Kabira

# *Language Reference*

**Kabira Infrastructure Server**

# Language Reference : Kabira Infrastructure Server

Published 14 May 2010



This document is intended to address *your* needs. If you have any suggestions for how it might be improved, or if you have problems with this or any other Kabira documentation, please send e-mail to pubs@kabira.com. Thank you! Feedback about this document should include the reference KISLR-3.1.0 . Updated on 14 May 2010 .

# Contents

# List of Tables

# 1

# Lexical and syntactic fundamentals

This chapter describes the basic lexical rules of the Kabira Infrastructure Server (KIS) modeling language, IDLos. These rules follow the IDL (version 2.2) standard. These rules also apply to action language and build specifications.

The chapter covers the character set, tokens, white space, comments, pre-processing directives, and the syntax for properties.

## Character set

IDLos supports the ISO Latin-1 (8859.1) character set. This consists of:

### Decimal digits

These are: 0 1 2 3 4 5 6 7 8 9

### Graphic characters

The graphic characters are: ! " # $ % & ' ( ) * + = . / : ; < = > ? @ [ \ ] ^ _ ' { | } ~ , plus the extended set.

### Formatting characters

The formatting characters are: BEL, BS, HT, LF NL, VT, FF, CR.

### Alphabetic characters

The alphabetic characters are a-z and A-Z. plus these: àÀ áÁ âÂ ãÃ äÄ åÅ æÆ çÇ èÈ éÉ êÊ ëË ìÌ íÍ îÎ ïÏ ñÑ òÒ óÓ ôÔ õÕ öÖ øØ ùÙ úÚ ûÛ üÜ ÿ and the Icelandic thorn and eth characters, both capital and lowercase.

# Tokens

There are five kinds of tokens: keywords, identifiers, literals, operators and other separators.

## Keywords

The following table lists all the keywords reserved by IDLos.

Although this is a large table, there are only about 20 statements that need to be defined in the IDLos reference pages. IDLos really is a small language; many of the statements use two or three keywords, and there are 14 basic types of keywords. The trigger statement alone uses 11 keywords.

| | | |
|---|---|---|
| abort | fixed | relationship |
| abstract | float | role |
| action | from | select |
| annotation | ignore | sequence |
| any | in | singleton |
| attribute | initialize | short |
| boolean | inout | signal |
| by | interface | stateset |
| cannothappen | key | string |
| case | local | struct |
| char | long | super |
| commit | module | switch |
| const | native | terminate |
| context | Object | to |
| create | octet | transition |
| createaccess | order | trigger |
| default | oneway | true |
| delete | out | TRUE |
| deleteaccess | package | typedef |
| double | post-get | unsigned |
| during | post-set | union |
| enum | pre-get | unrelate |
| entity | pre-set | upon |
| exception | raises | using |
| expose | readonly | virtual |
| extentaccess | refresh | void |
| false | recovery | wchar |
| FALSE | relate | with |
| finished | | wstring |

The keywords context, fixed are parsed, but are not used by KIS Design Center at this time.

# Identifiers

An identifier is an arbitrarily long sequence of alphabetic, digit, and underscore ("_") characters. The first character must be an alphabetic character. Only one of the letters A-Z or a-z may be used as the first character. Identifiers are case sensitive.

Unlike IDL, identifiers in IDLos are case sensitive.

If you need to use a keyword as an IDLos identifier, you can delimit it with quotation marks, for example:

```
entity purchase
{
    attribute long amount;
    attribute boolean "commit";
}
...
declare purchase p;
create p;
p."commit"=FALSE;
```

You can use any IDLos or action language keyword as a quoted IDLos identifier, unless it is also a C++ or IDL reserved word. Be aware that in some cases this can lead to ambiguity, for example:

```
void myAction(in long "commit"); // quoted keyword as parameter
...
                     // if the action contains this expression
if ("commit" == 1)   // then "commit" is parsed as a string literal
                     // and not as the input parameter
```

# Literals

Literals may be integer, character, floating point, string, or boolean:

| | |
|---|---|
| integer | A sequence of numerals beginning with 0 (e.g. 077) is interpreted as an octal number. A sequence of numerals beginning with any other number (e.g. 63) is interpreted as a decimal number. A sequence of numerals prefixed with 0x or 0X (e.g. 0x3F) is interpreted as a hexadecimal number. Hexadecimals may use a-f or A-F. A sequence of numerals appended with LL is interpreted as a 64-bit integer (long long). |
| character | A character literal is enclosed in single quotes (e.g. 'Z'). Standard IDL escapes are allowed, such as '\n' for newline and '\x67' for hexadecimal rendition. There is no support for wide character literals at this time. |
| floating point | A floating point literal consists of an integer part, a decimal point, a fraction part, and e or E, and an optionally signed integer exponent (e.g. 1.024+e13). The decimal point or the |

| | |
|---|---|
| | exponent may be missing, but not both. The integer part or the fraction part may be missing, but not both. |
| string | String values are placed in double quotes (e.g. "This is a string"). Adjoining strings are concatenated to form long strings. Newlines and other characters must be embedded with escape sequences. |
| boolean | A boolean literal: TRUE, true, FALSE, or false. |

# Escape sequences

The following table defines the escape sequences

| Description | Escape sequence |
|---|---|
| newline | \n |
| horizontal tab | \t |
| vertical tab | \v |
| backspace | \b |
| carriage return | \r |
| form feed | \f |
| alert | \a |
| backslash | \\ |
| question mark | \? |
| single quote | \' |
| double quote | \" |
| value of character in octal | \ooo |
| value of character in hexadecimal | \xhh |

# Operators

The operator tokens in IDLos are: - + ~ = ! && / * ( ) ?.

# Other separators

Other separators in IDLos are:

- preprocessor token #
- action language delimiter ‘
- statement delimiter ;
- list delimiter ,
- inheritance delimiter :
- scoping delimiter ::

- block grouping { }
- list grouping ( )
- array definition, property grouping [ ]
- sequence declaration < >
- comment tokens // /* */
- role multiplicity delimiter ..

# White space

White space is comprised of blanks, horizontal and vertical tabs, newlines, form feeds and comments. White space is ignored except as it serves to separate tokens. For example, when an inherited interface must be scoped

```
interface A : ::SomePackage::B {};    // correct
interface B :::SomePackage::B {};     // wrong! ::: causes error
```

# Comments

Single line comments start with // and continue to a newline. C-style comments start with a /* and continue until a */, and may include newlines.

Comments do not nest. Once a comment has started, all comment delimiters are ignored except the end delimiter for that style of comment.

# Preprocessing directives

The following preprocessing directives are supported. Preprocessing directives are not terminated by a semi-colon. Any other line where the first non-white space character is '#' is ignored.

| | |
|---|---|
| #include <filename> | When #include is the first non-white space on a line, the text after the #include is interpreted as a file name and will be included at that point in the text. The file is processed as IDLos. |
| #pragma include <filename> | Include the specified file in the generated output. The file is not parsed by the IDLos loader. |
| #pragma inline {`...`}; | Defines C++ code that is inlined with each generated implementation file for a component. The code within the enclosing backticks is not parsed, but directly included at the beginning of the generated C++ implementation files.<br><br>Since this code is inlined, only static functions should be used. Otherwise duplicate symbols error will result if multiple implementation files are generated. |

| | |
|---|---|
| #pragma header {`...`}; | Defines C++ code that is added to the generated header file for a component. The code within the enclosing backticks is not parsed, but added to the end of the generated C++ public header file for the component. |
| #pragma cpp {`...`}; | Defines C++ code that is added to the public library (or "stub library") for a component. The code within the enclosing backticks is not parsed, but added to the beginning of the generated stub C++ modules for the component. |
| #pragma | All other #pragma statements are not parsed. They are loaded into the repository and exported as-is by the exporter. This provides round-trip support for #pragma statements. |

KIS IDLos supports #include “headerfile.h”. The IDLos loader adds the local directory "." to the include path so there is no difference between #include <headerfile.h> and #include "headerfile.h". This differs from normal cpp rules for #include.

The supported `#pragmas` must be defined within a package. But the code is not part of the package C++ namespace - instead, it becomes part of the package's generated header, stub and implementation files.

The order of `#pragma` of each type is preserved. The order for different types is not preserved.

There is no way to debug `#pragma`-included code in the Kabira Model Debugger.

There is no mapping between C++ compiler error in the `#pragma`-included code and the idlOS file. You will need to examine the generated C++ files to determine where such errors originate.

Because of these restrictions, `#pragma`-included code should only be used if the code cannot be used in idlOS actions. Typical use cases would be interfacing to C++ libraries that need users to define subclasses, or C libraries that require function callbacks. In such cases you would define the class using `#pragma header`, and define the implementation using `#pragma cpp`.

# Quoted Keywords

IDLos allows keywords to be escaped by putting the identifier in double quotes. This allows reserved IDLos keywords to be used as normal identifiers if required.

For example, the following is legal IDLos:

```
//
//     Escape the reserved words interface and entity
//
interface "interface" { };
entity "entity" { };
expose entity "entity" with interface "interface";
```

# 2

# IDLos types

This chapter describes the Kabira Infrastructure Server (KIS) type system.

Types are labels that specify what category of information a symbol indicates. Usually one category is not comparable with another (for example, a string is not comparable with a float). KIS uses the IDL (version 2.2) type system, which provides a rich system of basic and user-defined types.

For each type, there is a brief description of the semantics of the type, and, where appropriate, how to specify the type in IDLos and/or Action Language. That will be followed by one or more examples, and any general information.

The following types are supported by KIS. Simple types are marked with an asterix(*).

- the section called “any” on page 8*the section called “array” on page 9, the section called “boolean” on page 10*, the section called “bounded sequence” on page 10
- the section called “bounded string” on page 12, the section called “bounded wstring” on page 12, the section called “char” on page 13*
- the section called “const” on page 13, the section called “double” on page 14*, the section called “entity” on page 15, the section called “enum” on page 15
- the section called “exception” on page 16,
- the section called “long” on page 19*, the section called “long long” on page 19*, the section called “ native” on page 20
- the section called “Object” on page 21*, the section called “octet” on page 21*, the section called “sequence” on page 22, the section called “short” on page 23*
- the section called “string” on page 24*, the section called “struct” on page 24, the section called “typedef” on page 26, the section called “union” on page 27, the section called “unsigned long” on page 28*

- the section called “unsigned long long” on page 29*, the section called “unsigned short” on page 30*
- the section called “void” on page 30, the section called “wchar” on page 31*, the section called “wstring” on page 31

# any

The type any is an IDL simple type. It can hold any IDLos type. An any logically contains a TypeCode, and a value that is described by the TypeCode.

## IDLos syntax

Inside of an entity block, define an attribute “name” of type any with:

```
attribute any name;
```

Define an array “name” of type any with:

```
typedef any name[5];
```

Define a sequence “name” of type any with:

```
typedef sequence<any> name;
```

Define a bounded sequence “name” of type any with:

```
typedef sequence<any, 5> name;
```

## Action language syntax

Inside of an action language function, allocate a handle “name” of type any with:

```
declare any name;
```

Inside of an action language function, allocate an array “name” of type any with:

```
declare any name[5];
```

## Example

```
declare any myAny;
declare any myAny2;
declare string str;
declare string str2;
str = "A string lives here.";
myAny <<= str;
myAny2 = myAny;
myAny2 >>= str2;
```

## General Information

Operator Descriptions:

| | |
|---|---|
| equals; | Assigns one any to another any |
| <<= | Inserts a value into an any. |

| | |
|---|---|
| >>= | Extracts a value from an any. |

The any is always the first operand of the insertion and extraction operators.

If there is a type mismatch between the value contained in an any and the target of an extraction operator, the following exception occurs:

```
swbuiltin::ExceptionAnyTypeMismatch
```

You should wrap an extraction operation in a try-catch block that catches the above exception.

```
action myPackage::myEntity::myOp
{`
     declare any myAny;
     declare string str;
     str = "A string lives here.";
     myAny <<= str;
     ...
     try
     {
         myAny >>= str;
         printf("%s\n", str.getCString());
     }
     catch (swbuiltin::ExceptionAnyTypeMismatch)
     {
         printf("A string doesn't live here any more.\n");
     }
`};
```

# array

The type  array is an IDL composite type. It represents a known-length series of a single type. An array must be named by a typedef declaration in order to be used as a parameter, or a return value. You can omit a typedef declaration only for an array that is declared within a structure or attribute definition.

## IDLos syntax

See IDLos syntax for a simple type, such as any.

## Action language syntax

See Action language syntax for a simple type, such as any.

## Example

This struct defines 256x256 array of 8-bit octets to hold an image:

```
struct SmallImage
{
     octet pixel[256][256];
};
```

## General Information

IDLos and Action Language support any number of dimensions. All dimensions in an array must be bounded. To define an array with unbounded dimensions, use a sequence.

# boolean

The type boolean is an IDL simple type. It represents a data item that can only be assigned the values of true or false.

## IDLos syntax

Inside of an entity block, define an attribute “name” of type boolean with:

```
attribute boolean name;
```

Define an array “name” of type boolean with:

```
typedef boolean name[5];
```

Define a sequence “name” of type boolean with:

```
typedef sequence<boolean> name;
```

Define a bounded sequence “name” of type boolean with:

```
typedef sequence<boolean, 5> name;
```

## Action language syntax

Inside of an action language function, allocate a handle “name” of type boolean with:

```
declare boolean name;
```

Inside of an action language function, allocate an array “name” of type boolean with:

```
declare boolean name[5];
```

## Example

```
declare boolean result;
result = true;
```

# bounded sequence

The type bounded sequence is an IDL template type. Bounded sequences can hold any number of elements, up to the limit specified by the bound. A bounded sequence must be named by a typedef declaration in order to be used as a parameter, an attribute, or a return value. You can omit a typedef declaration only for a sequence that is declared within a structure definition.

Bounded sequences have a special `length` attribute that you use in action language to determine the current number of elements in the sequence (see the action language example below).

# IDLos syntax

```
typedef sequence<any,5> name;
```

# Action language syntax

Sequences cannot be defined other than as a typedef or as a member of a struct or exception.

# Example

```
const long constBoundsValue = 50;
typedef string strArray[constBoundsValue];
typedef sequence<strArray,constBoundsValue> boundedSeqOfStrArrys;
struct Account
{
    string customer;
    string cardCompany;
    sequence <unsigned short, 10> creditCard;
};
struct LimitedAccounts {
    string<10> bankSortCode;
    sequence<Account, 50> accounts; // max sequence length is 50
};
```

# Action language example

```
package test
{
   const long constBoundsValue = 10;
   typedef sequence<string, constBoundsValue> IndexList;

   [local]
   entity testEntity
   {
      [initialize]
      void doIt();
      action doIt
      {`
         declare long long i;
         declare IndexList myList;
         for (i=1;i<=constBoundsValue;i++)
         {
            myList[myList.length] = i;
         }
         myList[5] = 5;
         printf("List values: ");
         for (i=0;i<constBoundsValue;i++)
         {
            printf(myList[i]);
            printf(" ");
         }
      `};
   };
};
```

The component outputs the following line to `coordinator.stdout` :

```
List values: 1 2 3 4 5 5 7 8 9 10
```

# bounded string

The type  bounded string is an IDL template type. It is a series of any number of possible 8-bit quantities, up to the limit specified by the bound.

## IDLos syntax

You may define a bounded string anywhere you can define a simple type.

```
attribute string<1024> buffer;
```

## Action language syntax

```
declare string<10> myString;
```

## Example

```
struct ShortMessagePacket
 {
     string destination;
     string<1024> shortMessage;
 };
```

# bounded wstring

The type  bounded wstring is an IDL template type.

## IDLos syntax

You may define a bounded wstring anywhere you can use a simple type.

```
 struct ShortMessagePacket
 {
     wstring destination;
     wstring<1024> shortMessage;
 };
```

## Action language syntax

```
declare wstring<10> wideLabel;
```

For string assignments and implicit string conversion rules, see “Manipulating data” on page 104.

## Example

```
struct ShortMessagePacket;
{
    wstring destination;
    wstring<1024> shortMessage;
};
declare wstring<1024> wideBuffer;
```

# char

The type  char is an IDL simple type. It is an 8-bit quantity that encodes a single-byte character from any byte-oriented code set. The type char can hold any value from the ISO Latin-1 character set. Code positions 0-127 are identical to ASCII. Code positions 128-255 are reserved for special characters in various European languages, such as accented vowels.

## IDLos syntax

Inside of an entity block, define an attribute “name” of type char with:

```
attribute char name;
```

Define an array “name” of type char with:

```
typedef char name[5];
```

Define a sequence “name” of type char with:

```
typedef sequence<char> name;
```

Define a bounded sequence “name” of type char with:

```
typedef sequence<char, 5> name;
```

## Action language syntax

Inside of an action language function, allocate a handle “name” of type char with:

```
declare char name;
```

Inside of an action language function, allocate an array “name” of type char with:

```
declare char name[5];
```

## Example

```
declare char name;
```

# const

Used in conjunction with other IDL types,  const assigns a symbol to an unchanging value.

These types can be made const: boolean, char, double, float, long, long long, short, string, unsigned long, unsigned long long, unsigned short, wchar, wstring.

## IDLos syntax

```
entity name
{
    const short parm = 5;
};
```

## Example

```
const long numOfEnglishLetters = 26;
typedef string EnglishIndexURLs[numOfEnglishLetters];
entity htmlIndex
{
      attribute EnglishIndexURLs URLs;
     boolean getIndexURL(in long i, out string url);
     action getIndexURL
     {`
         if (i >= 0 && i < numOfEnglishLetters)
         {
             url = self.URLs[i];
             return true;
         }
         else
         {
             url = "";
             return false;
         }
     `};
};
```

## General Information

There is no such thing as an any constant value.

A constant value can be specified as a literal value or a scoped name to another user defined constant value.

# double

The type double is an IDL simple type. It represents IEEE double-precision floating point numbers. Types float and double follow IEEE specifications for single- and double-precision floating point values, and on most platforms map to native IEEE floating point types.

## IDLos syntax

Inside of an entity block, define an attribute “name” of type double with:

```
attribute double name;
```

Define an array “name” of type double with:

```
typedef double name[5];
```

Define a sequence “name” of type double with:

```
typedef sequence<double> name;
```

Define a bounded sequence “name” of type double with:

```
typedef sequence<double, 5> name;
```

## Action language syntax

Inside of an action language function, allocate a handle “name” of type double with:

```
declare double name;
```

Inside of an action language function, allocate an array “name” of type double with:

```
declare double name[5];
```

## Example

```
declare double name;
```

# entity

The type  entity is an IDL composite type.

With an entity, you can:

- define attributes and operations
- inherit from other entities
- define any number of relationships with other entities
- make it accessible to other packages using an interface
- trigger operations when it is created, deleted, refreshed, updated, or related to another entity
- define keys to use in queries
- manage its states with a finite state machine
- define types nested in the entity

## IDLos syntax

```
 entity name ( : <parent type > )
 {
        …
}
```

## Example

```
entity TimerEventImpl
{
     // attributes
     attribute Road road;
     // operations
     oneway void generate ();
};
```

# enum

The type  enum is an IDL composite type.

An enum (enumerated) type lets you assign identifiers to the members of a set of values.

## IDLos syntax

```
enum name { <list of values ( = const_expr )>;
```

## Example

```
enum SideDirection {North, East = 5, South, West};
```

This defines SideDirection as a type which takes on the values North, East, South or West.

Action Language example:

```
declare enum SideDirection localDirection;
localDirection = East;
```

## General Information

The value of an enum is not defined by IDLos other than IDLos guarantees that the ordinal values of enumerated types monotonically increase from left to right. The value of an enum can also be explicitly defined by the modeler using an initializer value. All enumerators are mapped to a 32-bit type.

# exception

The type exception is an IDL composite type.

The IDLos exception type itself is similar to a struct type; it can contain various data members (but no methods). An exception can inherit from another exception. All attributes in the parent type are available in the child type.

An exception is used to alter the flow of control of a section of code. When a method raises an exception, the method "returns" at that point in its code. When the calling code receives the exception, control passes to any matching catch blocks, or the exception is rethrown.

Uncaught user and system exceptions will cause a runtime engine failure.

## IDLos syntax

```
exception name ( : < parent type > ) { <attribute list> };
```

## Example

```
exception IsDead { string causeOfDeath; };
void wakeup() raises (IsDead);
```

This defines the user exception IsDead as a type with attribute causeOf Death. The function wakeup() can instantiate a variable of type IsDead, set the value of causeOfDeath and throw the variable as a user exception. To catch the above exception:

```
try { self.wakeup(); } catch (IsDead id) {...}
```

## General Information

User exceptions are defined in IDLos.

System exceptions are defined in swbuiltin.sdl. They can be raised by the KIS runtime and adapters. The user should not throw System exceptions, but should catch some of them in certain conditions. To catch system exceptions in action language, include swbuiltin.sdl in the model.

Handling the system exceptions ExceptionDeadlock and ExceptionObjectDestroyed in application code not executed as part of spawn will produce unpredictable but bad results.

Operations that throw an exception must have a raises clause in the operation signature.

An exception definition cannot include itself as a member.

# float

The type  float is an IDL simple type.

It represents IEEE single-precision floating point numbers. Types float and double follow IEEE specifications for single- and double-precision floating point values, and on most platforms map to native IEEE floating point types.

## IDLos syntax

Inside of an entity block, define an attribute “name” of type float with:

```
attribute float name;
```

Define an array “name” of type float with:

```
typedef float name[<size>];
```

Inside of an entity block, define a sequence “name” of type float with:

```
typedef sequence<float> name;
```

## Action language syntax

Inside of an action language function, allocate a handle “name” of type float with:

```
declare float name;
```

Inside of an action language function, allocate an array “name” of type float with:

```
declare float name[<size>];
```

## Example

```
declare float name;
```

# interface

An IDLos interface exposes an entity's operations and interfaces. An interface is the mechanism to make implementation visible outside of package boundaries.

An IDL attribute is short-hand for a pair of operations that get and, optionally, set values in an object.

# IDLos syntax

```
interface Employee
{
     // only these parts of EmployeeImpl will be
     // accessible outside the package.
     void promote();
     oneway void makewaves();
     attribute string name;
};
entity EmployeeImpl
{
     void promote();
     attribute string name;
     attribute string m_nickname;
     signal makewaves();
};
```

# Example

In this example, the EmployeeImpl entity has two attributes, an operation, and a signal. The interface exposes all but the m_nickname attribute. Notice that the signal is exposed as an operation.

```
interface Employee
{
     // only these parts of EmployeeImpl will be
     // accessible outside the package.
     void promote();
     oneway void makewaves();
     attribute string name;
};
entity EmployeeImpl
{
     void promote();
     attribute string name;
     attribute string m_nickname;
     signal makewaves();
};
// This statement is self-explanatory.
expose entity EmployeeImpl with interface Employee;
```

# General Information

An interface may also contain definitions for typedefs, attributes, operations, enumerations, and structures.

Each interface may expose only one entity. An entity may be exposed by more than one interface. Each operation or attribute in the interface must have a corresponding attribute, operation or signal in the entity.

Abstract interfaces may not contain attributes or typedefs.

To permit more control over how an entity is exposed, three properties grant or revoke access to create, delete, or retrieve the extent of an interface.

# long

The type  long is an IDL simple type. It represents 32-bit signed integer values. IDL supports short and long integer types, both signed and unsigned. IDL guarantees the range of these types.

## IDLos syntax

Inside of an entity block, define an attribute “name” of type long with:

```
attribute long name;
```

Define an array “name” of type long with:

```
typedef long name[size];
```

Define a sequence “name” of type long with:

```
typedef sequence<long> name;
```

## Action language syntax

Inside of an action language function, allocate a handle “name” of type long with:

```
declare long name;
```

Inside of an action language function, allocate an array “name” of type long with:

```
declare long name[<size>];
```

## Example

```
declare long name;
```

## Warnings

If you do not `declare` a long inside of an action language function, the model compiler will use the native C type "long", which on many platforms is a 64 bit integer.

```
declare long idlosLong; // 32-bit
long nativeLong; // may be 64-bit
```

It is best to declare all variables, to avoid any uncertainty about type sizes.

# long long

The type  long long is an IDL simple type.

The long long type represents 64-bit signed integer values.

## IDLos syntax

Inside of an entity block, define an attribute “name” of type long long with:

```
attribute long long name;
```

Define an array “name” of type long long with:

```
typedef long long name[<size>];
```

Define a sequence “name” of type long long with:

```
typedef sequence<long long> name;
```

## Action language syntax

Inside of an action language function, allocate a handle “name” of type long long with:

```
declare long long name;
```

Inside of an action language function, allocate an array “name” of type long long with:

```
declare long long name[<size>];
```

## Example

```
declare long long name;
```

# native

A native type is used to represent a platform specific type that is opaque to the IDLos type system. The only Design Center audits performed on a native are to ensure that it does not have an initial value specified and that it is not used in a modeled const definition.

The native type can be specified using a local or global scope. See the example below.

## IDLos syntax

Inside of an entity block, define an attribute “name” of type native with:

```
native <native type> name;
```

## Action language syntax

Inside of an action language function, allocate a handle “name” of type native with:

```
declare <native type name> name;
```

## Example

```
package mypackage
{
   //
   //    This native is in the local package name space
   //
   native LocalType     LocalNative;

   //
   //    This native is in the global name space
   //
   native ::GlobalType GlobalNative;
```

```
};

//
//     Using the native types in action language
//
declare LocalNative     myLocalNative;

declare GlobalNative    myGlobalNative;
```

# Object

The type  Object is an IDL simple type. It represents a reference to a user-defined interface or entity.

## IDLos syntax

Inside of an entity block, define an attribute “name” of type Object with:

```
attribute Object name;
```

## Action language syntax

Inside of an action language function, allocate a handle “name” of type Object with:

```
declare Object name;
```

## Example

```
declare Object name;
```

# octet

The type  octet is an IDL simple type.

It represents an 8-bit quantity guaranteed not to undergo any conversion when transmitted by the communication system. This lets you safely transmit binary data between different address spaces. Avoid using type char for binary data, inasmuch as characters might be subject to translation during transmission. For example, if client that uses ASCII sends a string to a server that uses EBCDIC, the sender and receiver are liable to have different binary values for the string's characters.

## IDLos syntax

Inside of an entity block, define an attribute “name” of type octet with:

```
attribute octet name;
```

Inside of an entity block, define a sequence “name” of type octet with:

```
typedef sequence<octet> name;
```

## Action language syntax

Inside of an action language function, allocate a handle “name” of type octet with:

```
declare octet name;
```

Inside of an action language function, allocate an array “name” of type octet with:

```
declare octet name[<size>];
```

## Example

```
declare octet name;
```

# sequence

The type sequence is an IDL template type. It represents an array with unbounded dimensions. Unbounded sequences are automatically grown at runtime as required. If the index being accessed is larger than the current sequence size, the sequence is resized. When a sequence is resized, the values of all new sequence members are undefined. They must be initialized to a known value by the application.

The index for sequences starts at 0. You must use long long or unsigned long long types when iterating over a sequence, otherwise a compilation warning may be generated.

A sequence must be named by a typedef declaration in order to be used as a parameter, an attribute, or a return value. You can omit a typedef declaration only for a sequence that is declared within a structure definition.

Sequences have a special `length` attribute that you use in action language to determine the current number of elements in the sequence (see the action language example below). The sequence length is an unsigned long long value.

If the maximum size of a sequence is known, you can also use a the section called “bounded sequence” on page 10.

## IDLos syntax

```
typedef sequence<string> NameList;
```

## Example

```
const long constBoundsValue = 50;
typedef string stringArray[constBoundsValue];
typedef sequence<stringArray> SeqOfStringArrays;
struct Account
{
    string customer;
    string cardCompany;
    sequence<unsigned short> creditCard;
};
struct UnlimitedAccounts
{
    string<10> bankSortCode;
    sequence<Account> accounts; // no max sequence length
};
```

## Action language example

Assume you start an engine implemented as follows:

```
package test1
{
```

```
   typedef sequence<string> IndexList;
   const long constBoundsValue = 10;

   [local]
   entity testEntity
   {
      [initialize]
      void doIt();
      action doIt
      {`
         declare long long i;
         declare IndexList myList;
         for (i=1;i<=constBoundsValue;i++)
         {
            myList[myList.length] = i;
         }
         myList[5] = 5;
         printf("List values: ");
         for (i=0;i<constBoundsValue;i++)
         {
            printf(myList[i]);
            printf(" ");
         }
      `};
   };
};
```

The component outputs the following line to `coordinator.stdout` :

```
List values: 1 2 3 4 5 5 7 8 9 10
```

## General Information

Sequences cannot be defined other than as a typedef or as a member of a struct or exception. An attribute length represents the current length of the sequence at runtime.

# short

The type short is an IDL simple type. It represents 16-bit signed integer values.

## IDLos syntax

Inside of an entity block, define an attribute “name” of type short with:

```
attribute short name;
```

Define an array “name” of type short with:

```
typedef short name[<size>];
```

Define a sequence “name” of type short with:

```
typedef sequence<short> name;
```

## Action language syntax

Inside of an action language function, allocate a handle “name” of type short with:

```
declare short name;
```

Inside of an action language function, allocate an array “name” of type short with:

```
declare short name[<size>];
```

## Example

```
declare short name;
```

# string

The type  string is an IDL template type. It is a series of all possible 8-bit quantities. Unbounded string lengths are generally constrained only by memory limitations.

You can provide allocation hints to the model compiler when declaring unbounded strings, by using the `sizehint` property on a `typedef string` statement. The model compiler allocates the requested size on the stack, which can improve performance over allocating memory in the heap to grow an unbounded string.

If the maximum size of a string is known, you can also use a the section called “bounded string” on page 12.

## IDLos syntax

```
attribute string name;
```

## Action language syntax

```
declare string name;
```

For string assignments and implicit string conversion rules, see “Manipulating data” on page 104.

## Example

```
typedef string myString;
[ sizehint=250000 ]
typedef string MyPresizedString;

attribute myString shortName;
attribute myPresizedString anotherName;
```

## General Information

An attribute length represents the current length of the string at runtime.

# struct

The type  struct is an IDL composite type. A struct data type lets you encapsulate a set of named members of various types.

# IDLos syntax

```
struct AStruct
{
     long    aLongMember;
     short    aShortMember;
};
```

# Action language syntax

```
declare AStruct    aStruct;
aStruct.aLongMember = 76543;
```

# Example

```
struct Packet
{
    long destination;
    octet data[256];
    boolean retry;
};
```

This defines Packet as a type with members destination, data and retry.

For structs, you can define the types of their members in-line.

```
struct A
{
     boolean firstMember;
     struct B
     {
         unsigned long a;
         long b;
         string c;
     } secondMember;
     long thirdMember;
};
```

You can also use standalone type definitions for enums, unions, structs, and const values in structures.

```
struct S
{
    // Const definition
    //
    const long SomeLong = 42;

    // Inlined type definition
    //
    enum X
    {
    X1,
    X2
    } x1;

    // Can use Inlined type definition after it is defined.
    //
    X x2;

    // Standalone type definition
    //
    enum Y
```

```
    {
    Y1,
    Y2
    };

    // Use of standalone type definition.
    //
    Y y;
};
```

## General Information

Because a struct provides a naming scope, member names must be unique only within the enclosing structure.

Structures with no fields are supported:

```
struct Base
{
    long    l;
};

struct Child : Base
{
};
```

Recursive definitions of a struct are not allowed:

```
struct nodeList
{
    string name;
    typedef sequence<nodeList> NextNode;  // Not allowed, fails audit
    NextNode nextNode;
};
```

A workaround is available using `sequence`:

```
struct nodeList
{
    string name;
    sequence<nodeList,1> nextNode;
};
```

# typedef

A typedef lets you define a new legal type; The typedef keyword allows you define a meaningful or simpler name for an IDL type.

## IDLos syntax

```
typedef sequence<unsigned long> name;
```

## Example

```
typedef enum status {started, finished} racerStatus;
```

## General Information

You may rename any legal type. In a typedef, you can define a sequence, struct, or enum in-line.

# union

The type  union is an IDL composite type. A union type lets you define a structure that can contain only one of several alternative members at any given time. All IDL unions are discriminated.

## IDLos syntax

You declare a union type with the following IDL syntax:

```
union name switch (discriminator)
{
    case label1 : element-spec;
    case label2 : element-spec;
    [...]
    case labeln : element-spec;
    [default : element-spec;]
};
```

## Action language syntax

```
union MyUnion switch (char)
```

## Example

You can access the value of the discriminator in action language using the special _d attribute:

```
union MyUnion switch (char)
{
 case 'S':
 case 's':
     string  aString;
 case 'L':
 case 'l':
     long    aLong;
  default:
     long    justANumber;
};

enum Direction
{
    North,
    South,
    East,
    West,
};
union DirectionUnion switch (Direction)
{
    case South:
    case North:
        long latitude;
    case East:
    case West:
        long longitude;
};
struct ComplexStructure
{
```

```
   string name;
   MyUnion myUnion;
};
...
declare MyUnion mu;
mu.aString = ”some data”;
...
if (mu._d == 'S'    mu._d == 's')
{
     // it's a string
}
```

You can also set the discriminator...

```
mu._d = 'L';
```

...but KIS does not check that you are setting the discriminator to a legitimate value.

You can also set the union to null...

```
declare DirectionUnion geoLine;
geoLine = isnull;
```

To handle a union in a complex structure, you must declare a local union to set and then move it as a whole into the structure...

```
declare ComplexStruct myStruct;
declare MyUnion mu;
myStruct.name = “Joe”;
mu.aLong;
myStruct.myUnion = mu;
```

## General Information

The union is stored at runtime as an array of slots for each union member. This storage is not optimized; the runtime size of a union is simply the sum of the union members.

The above means that unions are not the best choice for systems where small size, or high performance are important. Avoid using unions as a placeholder for incomplete analysis; try to use them only when they are the best solution to a problem.

The Kabira Monitor displays the discriminator and the current values of all the union members.

If you access an “incorrect” union member (that is, the discriminator indicates another union member is active), the accessor throws an ExceptionDataError system exception.

A struct, union, or exception definition cannot include itself as a member.

A union's discriminator can be integer, char, boolean or enum, or an alias of one of these types; all case label expressions must be compatible with this type. Because a union provides a naming scope, member names must be unique only within the enclosing union. Unions allow multiple case labels for a single member. Unions can optionally contain a default case label. The corresponding member is active if the discriminator value does not correspond to any other label.

# unsigned long

The type  unsigned long is an IDL simple type. It represents 32-bit unsigned integer values.

## IDLos syntax

Inside of an entity block, define an attribute “name” of type unsigned long with:

```
attribute unsigned long name;
```

Define an array “name” of type unsigned long with:

```
typedef unsigned long name[<size>];
```

Define a sequence “name” of type unsigned long with:

```
typedef sequence<unsigned long> name;
```

## Action language syntax

Inside of an action language function, allocate a handle “name” of type unsigned long with:

```
declare unsigned long name;
```

Inside of an action language function, allocate an array “name” of type unsigned long with:

```
declare unsigned long name[<size>];
```

## Example

```
declare unsigned long name;
```

# unsigned long long

The type  unsigned long long is an IDL simple type. It represents 64-bit unsigned integer values.

## IDLos syntax

Inside of an entity block, define an attribute “name” of type unsigned long long with:

```
attribute unsigned long long name;
```

Define an array attribute “name” of type unsigned long long with:

```
typedef unsigned long long name[<size>];
```

Define a sequence “name” of type unsigned long long with:

```
typedef sequence<unsigned long long> name;
```

## Action language syntax

Inside of an action language function, allocate a handle “name” of type unsigned long long with:

```
declare unsigned long long name;
```

Inside of an action language function, allocate an array “name” of type unsigned long long with:

```
declare unsigned long long name[<size>];
```

### Example

```
declare unsigned long long name;
```

# unsigned short

The type  unsigned short is an IDL simple type. It represents 16-bit unsigned integer values.

## IDLos syntax

Inside of an entity block, define an attribute “name” of type unsigned short with:

```
attribute unsigned short name;
```

Define an array attribute “name” of type unsigned short with:

```
typedef unsigned short name[<size>];
```

Define a sequence “name” of type unsigned short with:

```
typedef sequence<unsigned short> name;
```

## Action language syntax

Inside of an action language function, allocate a handle “name” of type unsigned short with:

```
declare unsigned short name;
```

Inside of an action language function, allocate an array “name” of type unsigned short with:

```
declare unsigned short name[<size>];
```

## Example

```
declare unsigned short name;
```

# void

The type  void is an IDL simple type.

The void keyword is valid only in an operation. In an operation declaration, it may be used to indicate an operation that does not return a function result value.

## IDLos syntax

```
void myOperation(in myParm);
```

## Example

```
void myOperation(in myParm);
```

# wchar

The type  wchar is an IDL simple type. It encodes wide characters from any character set. The size of a wchar is platform-dependent.

## IDLos syntax

Inside of an entity block, define an attribute “name” of type wchar with:

```
attribute wchar name;
```

Define an array “name” of type wchar with:

```
typedef wchar name[<size>];
```

Define a sequence “name” of type wchar with:

```
typedef sequence<wchar> name;
```

## Action language syntax

Inside of an action language function, allocate a handle “name” of type wchar with:

```
declare wchar name;
```

Inside of an action language function, allocate an array “name” of type wchar with:

```
declare wchar name[<size>];
```

## Example

```
declare wchar name;
```

# wstring

The type  wstring is an IDL template type. It is the wide-character equivalent of type string. Like string-types, wstring types can be unbounded or bounded.

Since the KIS implementation of the string type also allows for multi-byte strings, there is little difference between string and wstring. The only current difference is that wstrings cannot be compared as greater-than or less-than other wstrings.

If the maximum size of a wstring is known, you can also use a the section called “bounded wstring” on page 12.

## IDLos syntax

Inside of an entity block, define an attribute “name” of type wstring with:

```
attribute wstring name;
```

Define an array “name” of type wstring with:

```
typedef wstring name[<size>];
```

where <size> is an integer and is the number of bytes allocated for the string, (which may not be the number of characters).

Define a sequence “name” of type wstring with:

```
typedef sequence<wstring> name;
```

# Action language syntax

Inside of an action language function, allocate a handle “name” of type wstring with:

```
declare wstring name;
```

Inside of an action language function, allocate an array “name” of type wstring with:

```
declare wstring name[<size>];
```

# Example

```
declare wstring name;
name = “a single-byte character string”;
```

There is no built in support for wide string literals.

# 3

# IDLos Reference

This chapter contains reference pages for each statement in the Kabira Infrastructure Server (KIS) component modeling language, IDLos, in alphabetical order. Each page describes the syntax of the statement, a discussion of usage, and an example.

The page for properties contains a table of all the IDLos properties.

## Understanding the notation

The syntax description for each statement or function uses the following conventions:

**bold** screen font indicates a literal item that appears exactly as shown

*italics* indicates a language element that is defined elsewhere

`plain text` indicates a feature of the notation, as follows:

( item ) parentheses group items together

( item )* the item in parentheses appears zero or more times

( item )+ the item in parentheses appears one or more times

{ item } items in braces are optional

( item1 | item2 ) the vertical bar indicates either item1 or item2 appears

### Common elements of the grammar

The following element of the IDLos grammar appear frequently throughout the reference pages.

| Grammar element | Meaning |
|---|---|
| identifier | any supported characters, digit, or underscore. Must start with a-z or A-Z. |
| scoped_name | {::} simple_declarator (:: simple_declarator)* |
| simple_declarator | identifier |

# Files and engines in IDLos

This section discusses how IDLos can be arranged into files and mapped to KIS engines.

## Files

Usually you will put one package in a file. But there is no restriction on how IDLos packages are arranged in files except that each package must be syntactically complete.

If a package name appears again in a model, IDLos reopens the package and adds information to it. In this way, a package might be divided up among many files. For example, you might have each module from that package defined in its own file.

Actions might be defined in separate files. This is similar to the arrangement of C++ implementations kept in different files than the class definitions.

The Design Center does not care how your IDLos files are arranged. Whether you have one file that #includes all the necessary files, or you load each file by hand - the Design Center can build your model as long as you have all the necessary packages loaded when you build your engine.

## Engines

You will usually put one package in a file and generate one engine from that package.

Packages are mapped to engines using the KIS Design Center. There is no restriction in the Design Center on how many packages can be built into one engine. You may compile as many packages into an engine as you want.

When you partition Kabira applications into different components, the package represents the smallest level of granularity you can use.

# action

Defines the behavior of a state or an operation.

## Syntax

```
returnValue
action scoped_name ( parameterList )
    raises ( exceptionList )

{‘actionLanguage‘};
```

## Description

The implementation of a state or action is defined in action language, between the back quotes ‘ . An action can be defined in an entity, a module, a package, or outside of a package. The action declaration may optional include the signature of the operation or state that it is implementing.

## Warnings

None

## Example

```
void
action ::Traffic::GUIProxyImpl::updateMode (
   in ModeType modeType,
   in RoadDirection modeDir)
//
{`
    // some action language
`};
```

## See also

the section called “ exception” on page 39 , the section called “ stateset” on page 51 .

# attribute

An object data member, and its accessors.

## Syntax

```
{ readonly } attribute param_type_spec simple_declarator { = literal|scoped_const } ;
```

## Description

In an entity, the attribute defines a data member, a set accessor and a get accessor. In an interface, the attribute exposes those same accessors. If the attribute is `readonly` in the interface, only the get accessor is exposed. If an attribute is `readonly` in an entity, then it can only be set using a `values` clause when an instance of the entity is created.

Attributes are inherited. To expose the attribute in the interface, it must match the attribute in the entity in both name and type.

Attributes cannot be structs or arrays. However, user-defined types defined as structs or arrays can be attributes.

Attributes may optionally define an initial value. This initial value may be an C/C++ style literal constant or a scoped name denoted by a leading `::`.

Initial values are only supported for fundamental types, `enums` and `swbuiltin::date`.

When an object is created, the values of its attributes without initial values are undefined.

An `attribute` statement may have the `javaaccess` property.

An attribute statement may have the `configured` property.

## Warnings

None

## Example

```
attribute long   timeGreen;
attribute LightColor  colorTarget;
attribute short   baseCount = 16;
attribute string  homeCity = "Peoria";
attribute long   ::mypackage::SomeLongConstant;
attribute swbuiltin::date startingDate = "2006-06-01 09:00:00";

[configured = optional ]
attribute string anOptionalConfigurationValue;
```

## See also

the section called “ trigger” on page 53

# const

Defines a constant value.

## Syntax

```
const const_type identifier = { = literal|scoped_const };
```

## Description

literal must be a valid literal type: integer, character, floating point, string, or boolean (see “Literals” on page 12 for more details).

*`scoped_const`* is a valid scoped name to another constant definition.

Assigning literal to an identifier of type *`const_type`* must represent a valid operation. `const_type` can be a scoped name. The following table shows valid IDLos types for each literal type:

| **literal** | **const_type** |
|---|---|
| integer | long long long |
| character | char wchar |
| floating point | float double |
| string | string wstring |
| boolean | boolean |

KIS also performs implicit conversions between string and numeric types in a `const` statement.

A `const` statement may have the `javaaccess` property.

## Warnings

A `const` may not be used as an array index unless it is defined outside the entity or interface in which it appears, for example at the package scope.

## Example

```
const long constBoundsValue = 50;
const char littleN = 'n';
const float smallFloat = 73.033e-32;
const string msg1233 = "Don't do that!";
const boolean SureThing = TRUE;
const boolean AnotherSureThing = SureThing;

// used as bound in string, array and sequence
typedef string<constBoundsValue> boundedString;
typedef string stringArray[constBoundsValue];
typedef sequence<string, constBoundsValue> stringSequence;
```

# enum

Defines a type whose values are a fixed set of tokens. Optionally defines specific integer values for the tokens.

## Syntax

```
enum identifier {identifier (= value(, identifier (= value)))*};
```

## Description

Enumerations contain a comma-separated list of one or more enumerators, which are just identifiers. These identifiers may optionally contain an assigned value, which may be either a literal constant integer value (a signed 32 quantity) or another scoped identifier.

An `enum` statement may have the `javaaccess` property.

## Warnings

None

## Example

```
enum ModeType
{
    tyTimed,
    tySensed,
    tyHoldGreen = 77,
    tyBlinkYellow,
    tyBlinkRed = lights::Red,
```

```
    tyNoChange
};
```

# entity

Defines the implementations of objects.

## Syntax

```
entity simple_declarator {: scoped_name} {(entityContent)*};
entity simple_declarator;
```

## Description

The first syntax is for entity definition, the second for forward declaration.

Entities can contain `actions`, attributes, enumerations, exceptions, keys, operations, signals, transitions, statesets, structs, triggers, and typedefs.

An entity statement may have `extentless`, `managedextent`, `annotation`, `local`, and `singleton` properties.

To make an entity accessible outside its package, expose the entity with an interface. An entity may be exposed by more than one interface.

The : *`scoped_name`* following the entity declarator designates a super type for inheritance. Entities may inherit from other entities if they are in the same package, and have the same value of their `local` property.

Entities may have relationships to other entities in the same package.

Forward declares allow you to use an entity type before it is defined.

## Warnings

Local entities have such a different description, they are covered on their own page.

An entity must be defined, not just forwarded, before it can be inherited from.

Multiple inheritance is not supported.

## Example

```
// forward declares
entity TimerEventImpl;

// definitions
entity TimerEventImpl
    {
    attribute Road road;
    oneway void generate ();
};
```

## See also

the section called “ local entity” on page 43 , the section called “ trigger” on page 53

# exception

A user-defined type that can be thrown and caught upon error.

## Syntax

```
exception identifier {: scoped_name} { (type_spec declarator;)*};
```

## Description

Define exceptions when you need to report errors in your applications.

The *scoped_name* following the exception declarator designates a super type for inheritance. Exceptions may inherit from other exceptions.

An `exception` statement may have the `javaaccess` property.

## Warnings

None

## Example

```
exception ExpFileNotFound
{
    string path;
    string filename;
};
```

## See also

the section called “operation” on page 47

action language “Exceptions” on page 163

# expose

Specifies which entity an interface exposes.

## Syntax

```
expose entity scoped_name with interface scoped_name;
```

## Description

An expose statement may have the `annotation` property. An abstract interface may not be used in an expose statement. You may define expose statements in a package or module.

## Warnings

None

## Example

```
expose entity GuiProxyImpl with interface GuiProxy;
```

## See also

the section called “ interface” on page 40 , the section called “ entity” on page 38

# interface

Interfaces provide access to entities for clients in other packages.

## Syntax

```
interface simple_declarator {: scoped_name} {(interface_body)*};
interface simple_declarator;
```

## Description

The first syntax is for interface definition. The second syntax is for forward declaration.

Interfaces can contain attributes, enums, exceptions, operations, structs, and typedefs.

Interfaces define what part of an entity gets exposed outside of the package boundary. Everything has open access within a package.

Interfaces contain the attributes and operations of the entity that you intend to expose to outside clients.

Signals are exposed as oneway void operations. They must be implemented in the entity.

The : scoped_name following the interface declarator designates a super type for inheritance. Interfaces may inherit from other interfaces, either within the same package or across package boundaries.

An `interface` statement may have `abstract` , `annotation` , `createaccess` , `deleteaccess` , `extentless` , `extentaccess`, `javaaccess`, `managedextent` and `singleton` properties.

You may also use interfaces for access to the entity from within the package; but it is not necessary.

## Warning

Relationships defined between interfaces are implemented between the entities they expose. Interfaces may not participate in associative relationships.

All attributes and operations listed in an interface must be defined in the exposed entity.

An interface must be defined, not just forwarded, before it can be inherited from.

## Example

```
interface Employee
{
// only these parts of EmployeeImpl will be
// accessible outside the package.
    void promote();
    oneway void makewaves();
    attribute string name;
};

entity EmployeeImpl
{
    void promote();
    attribute string name;
    attribute string m_nickname;
    signal makewaves();
};
expose entity EmployeeImpl with interface Employee;
```

## See also

the section called “ entity” on page 38 , the section called “ expose” on page 39

# key

A list of attributes whose values are used to select one or more instances of an entity.

## Syntax

```
key simple_declarator  {simple_declarator  (, simple_declarator)*};
```

## Description

Each key is a named list of attributes.

A key can contain as many attributes as you want. Multiple keys may be defined for a single entity.

Keys can be defined on both entities and roles in relationships. Keys defined on entities index all instances in the entity extent. Keys defined on roles only index the instances currently in the role.

Attempting to create an object with key values that are already part of a unique index will cause the runtime to throw a swbuiltin::ExceptionObjectNotUnique exception.

A `key` statement may have the `annotation, javaaccess, unique, or ordered` properties.

# Warnings

Arrays and sequences are not valid in keys.

When a key is inherited (that is, if an entity containing a key has subtypes), the index for the key is maintained in the type where the key is defined. This means that keys must be unique across instances of that type and all its subtypes.

Avoid modifying key attributes. If it is necessary to modify a attribute which is part of a key, enclose the modifications in a try..catch() block. This allows KIS to update the keys immediately after they are modified, and it lets you catch the `ExceptionObjectNotUnique` exception that can be thrown when you change a key. If you don't catch this exception, the engine will crash.

Here is an example of how you might use a temporary object when you need to change an attribute which is part of a key:

```
entity KeyedEntity
{
    attribute long longAttr;
    key Primary { longAttr };
};

. . . . . IDLos above, action language below . . . . .

declare KeyedEntity  keyedEntity;
declare long orgKey = 3;
declare long newKey = 4;

//
//  Always used a values clause when creating a keyed entity
//
create keyedEntity values (longAttr:orgKey);

//
//  Change the key
//
try
{
  keyedEntity.longAttr = newKey;
}
catch (swbuiltin::ExceptionObjectNotUnique)
{
  keyedEntity.longAttr = orgKey;
}
```

Key attributes should always be modified as a group within a `try` block.

If a key does not need to be modified after an object has been created the entity attribute (and any associated interface attribute) should be marked `readonly`. This allows the runtime to significantly optimize queries against that key value since it does not need to worry about key changes occurring once the index is created for an object. For example:

```
interface MyObject
{
     readonly attribute long id;

     key K { id };
```

```
};
entity MyObjectImpl
{
     readonly attribute long id;

     key K { id };
};
expose entity MyObjectImpl with interface MyObject;
```

The IDLos auditor will detect any attempts to modify this key at build time and fail to build the model.

## Example

```
entity Employee
{
    attribute string firstName;
    attribute string lastName;
    attribute long SSN;
    attribute long employeeNumber;
    key primary {employeeNumber};
    key secondary {SSN};
};


entity E
{
   attribute string s;
};

entity X
{
   attribute long l;
};

relationship R
{
   role E has 0..* X
   {
      key K { l };
   };
   role X back 0..* E
   {
      key K { s };
   };
};
};
```

## See also

the section called “ attribute” on page 35

action language `select , for ... in`

# local entity

An entity that cannot be distributed. A local entity is essentially a C++ class defined in IDLos and implemented in action language. Its operations are invoked directly without being dispatched through the KIS event bus, so they execute somewhat faster.

Though their operations are executed in a transaction, local entities are not transactional. This means that the invocation of operations in a local entity is not logged in the transaction log, and changes to the local entity are not logged in the transaction log, but everything that the local entity does to other entities is logged.

You specify a local entity by placing the `local` property before the entity declaration.

# Syntax

```
[local] entity simple_declarator {: scoped_name} {(entityContent)*};
entity simple_declarator;
```

The first syntax is for entity definition, the second for forward declaration.

# Description

Local entities can contain actions, enums, exceptions, operations, structs, and typedefs. To make a local entity accessible outside its package, expose the entity with an interface. A local entity may:

- be exposed by more than one interface
- inherit from other local entities in the same package
- use the `annotation` property
- have operations that use the `packageinitialize, packagereload, initialize , unload, reload` , and `terminate` properties

Local entities may not have relationships to other entities. Forward declares allow you to use a local entity type before it is defined.

Do not explicitly create or delete local entities. Declaring a local entity is sufficient to instantiate it.

# Warnings

A local entity cannot be used as an actual parameter when calling an operation on a non-local entity.

Local entities are not transactional.

Local entities have no state, so they cannot contain statesets.

# Example

```
// forward declare
entity StartUp;

[local]
entity StartUp
{
    [initialize]
    void initData ();
};
```

## See also

the section called “ entity” on page 38

# module

Used to provide extra name spaces within a package as needed.

## Syntax

```
module simple_declarator {(definition)+};
```

## Descriptions

Modules define a name space. Modules may be nested.

A module may contain entities, interfaces, enums, structs, exceptions, typedefs, nested modules, relationships, actions, and exposes statements. Modules are declared inside of packages.

A `module` statement may have the `annotation` property.

When a module name appears more than once in a model, information is added to the module definition each subsequent time that its name appears.

## Warnings

None

## Example

```
package Traffic
{
    module Pedestrians
    {
        entity Joggers {};
    };
    module Cars
    {
        entity Trucks {};
    };
    relationship
    {
        role Pedestrians::Joggers avoid 0..* Cars::Trucks;
        role Cars::Trucks stop 0..* Pedestrians::Joggers;
    };
};
```

# native

Used to alias a C or C++ type as an IDLos type

# Syntax

```
native {scoped_name} simple_declarator;
```

# Descriptions

Native is used to map an external type into the IDLos type system.

The optional scoped_name is used to specify the C or C++ type that should be mapped to the IDLos type name specified in simple_declarator.

The native type is defined in models using the simple_declarator name.

# Warnings

Native types are not marshaled over distribution. The value of a native type on a remote node is undefined.

Exposing native types through an interface is strongly discouraged. Native types are usually only valid in a single process. Exposing them in an interface makes the type potentially visible across engine boundaries.

# Example

```
native ::SWSMOffset     Offset;
native ::FILE     *FilePtr;
native ::CGObject     *ObjPtr;
typedef ObjPtr     ObjPtrTypeDef;

entity A
{
  attribute FilePtr    filePtr;
  attribute ObjPtr    objPtr;
  attribute ObjPtrTypeDef    objPtrTypeDef;
  attribute Offset    offset;
};

//
//   Using them in action language
//
a.filePtr = ::fopen("foo", "a");
if (::fclose(a.filePtr) != 0)
{
  declare swbuiltin::EngineServices engineServices;

  printf("FAILED: fclose failed %d\n", errno);
  engineServices.stop(1);
}

declare ObjPtr objPtr;

a.objPtr = new ::CGObject;
objPtr = a.objPtr;
delete objPtr;
a.objPtr = NULL;

a.objPtrTypeDef = new ::CGObject;
objPtr = a.objPtrTypeDef;
```

```
delete objPtr;
a.objPtrTypeDef = NULL;
```

# operation

Declares an operation on an entity or an interface.

## Syntax

```
{oneway}(param_type_spec   void) simple_declarator
                              ( { param_dcl (, param_dcl)*} )
            {raises (scoped_name (, scoped_name)*)};

param_dcl:     (in   out   inout) param_type_spec simple_declarator
```

## Description

Use `oneway` for asynchronous operations.

In an entity, an operation may have `annotation` , `const` , `inline, javaaccess, local` , `pure` , and `virtual` properties. The `pure` property may only be used together with `virtual` .

The `inline` property can only be used on operations that are not exposed by an interface, are not declared virtual, and are not used outside of the entity that defines the operation.

In an interface, an operation may have `annotation` , `const` , and `local` properties.

In a local entity, an operation may have `annotation` , `const` , `packageinitialize, packagereload, initialize` , `unload, load` , and `terminate` properties.

Operations are implemented in `action` statements.

## Warnings

Operations have constraints on whether they can be oneway, twoway, or have parameters when they:

- are used in triggers
- are used as engine events
- implement exposed signals

See the appropriate sections of the manual.

## Example

```
   void updateMode (
       in ModeType modeType,
       in RoadDirection modeDir);
   void updateLightTimes (
       in long timeGNS,
       in long timeGEW,
```

```
        in long timeYNS,
        in long timeYEW);
    void updateSensor (
        in boolean newCar,
        in SideDirection carDir);
    LightColor getLightN ();
    LightColor getLightS ();
    LightColor getLightE ();
    LightColor getLightW ();
    void getAllLights (
        out LightColor north,
        out LightColor south,
        out LightColor west,
        out LightColor east);
```

## See also

the section called “ action” on page 34 , the section called “ signal” on page 50 , the section called “ trigger” on page 53

# package

Packages define components.

## Syntax

```
package simple_declarator { (definition) * };
```

## Description

IDLos applications require packages. Packages define components. They form an access boundary: you cannot access an entity in another package unless it is exposed by an interface.

A package may contain entities, interfaces, enums, structs, exceptions, typedefs, modules, relationships, actions, and exposes statements.

A `package` statement may have the `annotation` property.

When a package name appears more than once in a model, information is added to the package definition each subsequent time that its name appears.

An IDLos file may contain more than one package, and a package may be contained in more than one file.

## Warnings

There is no implicit package scope for a file. All IDLos statements except `action` definitions must be encapsulated in an explicit package.

## Example

This fragment of the sample model has been edited for clarity’s sake.

```
package Traffic
```

```
{
    enum LightColor
    {
        tyRed,
        tyYellow,
        tyGreen,
        tyBlkRed,
        tyBlkYellow
    };
    interface GUIProxy
    {
        // operations
        LightColor GetLightN ();
        LightColor GetLightS ();
        LightColor GetLightE ();
        LightColor GetLightW ();
    };
};
```

# relationship / role

Defines a relationship between entities or between interfaces.

## Syntax

```
relationship simple_declarator
{
   role scoped_name identifier(0..1   0..*   1..1   1..*) scoped_name { { key identifier
 { identifier, ...} };
  {role scoped_name identifier(0..1   0..*   1..1   1..*) scoped_name { { key identifier
 { identifier, ...} }; }
  {using scoped_name;}
};
```

## Description

You specify roles in one or both directions between entities in the relationship statement. Each role has a name, a multiplicity, and specifies the two related entities. You can also specify an associative entity with the optional `using` clause.

You can specify a relationship between interfaces instead of between entities. This is how you expose a relationship. Do not mix entities and interfaces in a relationship. Interfaces may not be used as associative objects; relationships between interfaces may not contain a `using` clause.

Keys can be defined on roles. The key must use attributes in the to object of a role. Unique, non-unique, and ordered keys are supported.

A `relationship` statement may have the `annotation` property.

## Warnings

The two roles in a relationship must have different names.

## Example

```
entity E
{
```

```
    attribute string     s;
};

entity X
{
    attribute long     l;
};

relationship R
{
    role E has 0..* X
    {
        key K { l };
    };

    role X back 0..* E
    {
        key K { s };
    };

};
```

## See also

the section called “ trigger” on page 53

action language `relate` and `unrelate`

# signal

Defines an event for a state machine.

## Syntax

```
signal simple_declarator (param_dcl (, param_dcl)*);

param_dcl:
in param_type_spec simple_declarator
```

## Description

A `signal` statement may have the `annotation` property.

## Warnings

Signals may have only `in` parameters.

## Example

```
signal blinkRed( );
signal goRed( );
signal blinkYellow( );
signal greenTimed( );
signal greenSensor( );
signal greenHold( );
signal timeout( );
```

```
signal carArrived( );
signal goYellow( );
```

## See also

the section called “ stateset” on page 51 , the section called “ transition” on page 52 , the section called “operation” on page 47

# stateset

Defines the allowed states for an entity.

## Syntax

```
stateset {simple_declarator (, simple_declarator)*} = simple_declarator;
```

Where the last simple_declarator designates the initial state.

## Description

Defines the states in an entity’s state machine, and its initial state. When the entity is created, it is placed into its initial state.

Each state (including initial and final states) must be associated with an `action` . In KIS, the action of an initial state is not executed upon object creation. You must always transition into a state with a signal in order for an action to be executed.

A stateset statement may have `annotation` and `finished` properties.

## Warnings

State names are in the entity’s name space.

## Example

```
stateset
{
    Initial,
    BlinkRed,
    Red,
    TimedGreen,
    TimedYellow,
    SensorGreenPreferred,
    WaitForCar,
    WaitForGreenTimeout
} = Initial;
```

## See also

the section called “ action” on page 34 , the section called “ signal” on page 50 , the section called “ transition” on page 52

# struct

A type to hold data.

## Syntax

```
struct identifier {: scoped_name} { (type_spec declarator;)*};
```

## Description

Use structures as a container for data.

The *scoped_name* following the struct declarator designates a super type for inheritance. Structures may inherit from other structures.

A `struct` statement may have the `javaaccess` property.

A structure field may have the `configured` property.

## Warnings

None

## Example

```
struct Result
{
    typedef sequence<string> Messages;
    Messages messages;
    boolean status;
};
```

# transition

Defines the transition from one state to another when a specific signal is received.

## Syntax

```
transition scoped_name to (scoped_name   cannothappen   ignore)
           upon simple_declarator;
```

## Description

Transition statements specify how a state machine should respond to a certain signal when the entity is in a particular state. You can transition to another state, ignore the signal, or cause an exception to be thrown by specifying `cannothappen` . If a transition is not specified, the default is `cannothappen` .

## Warnings

There are no automatic transitions in IDLos. All transitions must be explicit.

## Example

## See also

the section called “ signal” on page 50 , the section called “ stateset” on page 51

# trigger

Triggers invoke operations upon the occurrence of some event.

## Syntax

```
trigger simple_declarator upon (create   delete   update | refresh
                                commit   abort);
trigger simple_declarator upon (pre-get   post-get   pre-set
                                post-set) simple_declarator;
trigger scoped_name upon ( relate   unrelate ) simple_declarator;
```

## Description

The first syntax is for entity triggers: The simple_declarator specifies an operation name. You define entity triggers in entities.

The second syntax is for attribute triggers. The first simple_declarator specifies an operation or signal. The next simple_declarator specifies the attribute upon which to trigger. These are also defined in entities.

The third syntax is for role triggers. The scoped_name identifies an operation or signal in the from entity of the role (the first one mentioned in the `role` statement). The simple_declarator specifies the `role` name. Role triggers are defined within relationship statements.

**Create, update, and delete triggers with inheritance** If you define create or update triggers for both the super type (parent) and subtype (child), then the super type trigger fires before the one in the subtype. Conversely, subtype delete triggers fire before those in a super type. Some readers will find this familiar, as it resembles class constructor/destructor behavior in other languages.

## Warnings

Operations used in triggers must return `void` and must have no parameters, and delete triggers may not be one-way operations. The operations must be defined in the entity for which the triggers are defined or in a super type.

### Example

```
trigger initializeAttributes upon create;
trigger calculatePayment upon pre-get mortgagePayment;
trigger deleteCustomer upon unrelate patronizes;
```

### See also

the section called " attribute" on page 35 , the section called "operation" on page 47 , the section called " signal" on page 50 , the section called " entity" on page 38 , the section called " relationship / role" on page 49

## typedef

Give a new name to a type. Define a new sequence or array type.

### Syntax

```
typedef type_spec declarator;
```

### Description

The declarator is the name of the new legal type. A declarator can include array dimensions.

The type_spec can be any legal type already defined, either a basic type, or a user-defined type. It could also be a sequence or a bounded string.

### Warnings

None

### Example

```
typedef octet byte;
typedef sequence<Employee> Division;
typedef boolean low_pass_filter[3][3];
```

## Complete IDLos grammar

```
accessValues  :
  granted     revoked

actionStatement  :
  action  scoped_name  swalBlock  ;

annotation  :
  annotation  =  string_literal  (  string_literal  )*

annotations  :
  [  annotation  ]

any_type  :
```

```
  any

array_declarator  :
 quoted_identifier  (  fixed_array_size  )+

assignedProperty  :
 annotation     finished  =  finishedStateList
 createaccess  =  accessValues    deleteaccess  =  accessValues
 extentaccess  =  accessValues  | javaaccess = javaAccessValues | sizehint = integer_type
|
 lockmode = lockMode | configured = configurationMode

assocSpec  :
  using  scoped_name

attr_dcl  :
  {  readonly  }  attribute  param_type_spec  simple_declarator
  {  =  string_literal  }
  (  ,  simple_declarator  {  =  string_literal  }  )*

attributeList  :
  {  simple_declarator  (  ,  simple_declarator  )*  }

attributeTrigger  :
  (  pre-set     post-set     pre-get     post-get  )
    simple_declarator

base_type_spec  :
  floating_pt_type     integer_type     char_type     wide_char_type
  boolean_type    octet_type     any_type     object_type

booleanProperty  :
  abstract  |  const  |  dynamiclog | extentless | initialize |
  inline | local  |  managedextent  | ordered |
  singleton  |  terminate  |  packageinitialize  |
  packagereload | pure  |  unique |
  unload | virtual

boolean_literal  :
  TRUE   true     FALSE   false

boolean_type  :
  boolean

case_clause  :
  (  case_label  )+  element_spec  ;

case_const_exp  :
  const_exp     unquoted_scoped_name

case_label  :
  case  case_const_exp  :      default  :

char_type  :
  char

character_literal  :
  character-literal

complex_declarator  :
  array_declarator

configurationMode  :
  optional     required

const_dcl  :
  const  const_type  simple_declarator  =  const_exp
```

```
const_exp  :
  {  (  -     +  )  }  literal

const_type  :
  integer_type     char_type     octet_type     wide_char_type
  boolean_type     floating_pt_type     string_type     wide_string_type
  fixed_pt_const_type    scoped_name

constr_type_spec  :
  struct_type     union_type     enum_type

context_expr  :
  context  (  string_literal  (  ,  string_literal  )*  )

declarator  :
  simple_declarator     complex_declarator

declarators  :
  declarator  (  ,  declarator  )*

definition  :
  type_dcl  ;     const_dcl  ;     except_dcl  ;     interface  ;
  module  ;     entity  ;    exposeStatement  ;
  relationshipStatement  ;     actionStatement     includeStatement
  pragmaStatement

element_spec  :
  type_spec  declarator

entity  :
  entity  (  entityStatement     forward_entity_dcl  )

entityBlock  :
  {  (  {  properties  }  entityContent  )*  }

entityContent  :
  signal_dcl  ;     outerStateSetStatement  ;
  transitionStatement  ;    entityTriggerStatement  ;
  keyStatement  ;     type_dcl  ;     const_dcl  ;     except_dcl  ;
  attr_dcl  ;     op_dcl  ;     actionStatement

entityStatement  :
  simple_declarator  {  inheritance_spec  }  entityBlock

entityTrigger  :
  (  create     delete     refresh     abort     commit
    state-conflict | update  )

entityTriggerStatement  :
  trigger  simple_declarator  upon
  (  attributeTrigger     entityTrigger  )

enum_type  :
  enum  simple_declarator  {  enumerator  (  ,  enumerator  )*  }

enumerator  :
  simple_declarator

except_dcl  :
  exception  simple_declarator  {  inheritance_spec  } {  (  except_member  )*  }

except_member  :
  type_spec  declarators  ;

exposeStatement  :
  expose  ( entity  scoped_name  with  interface  scoped_name  )
```

```
finishedStateList  :
  {  stateName  (  ,  stateName  )*  }

fixed_array_size  :
  [  template_bounds  ]

fixed_pt_const_type  :
  fixed

fixed_pt_type  :
  fixed  <  positive_int_const  ,  integer_literal  >

floating_point_literal  :
  floating-point-literal

floating_pt_type  :
  float     double     long  double

forward_dcl  :
  simple_declarator

forward_entity_dcl  :
  simple_declarator

includeStatement  :
  include  filename

inheritance_spec  :
  :  scoped_name  (  ,  scoped_name  )*

initState  :
  stateName

integer_literal  :
  decimal-integer-literal     hexadecimal-integer-literal     octal-integer-literal

integer_type  :
  signed_int     unsigned_int

interface  :
  interface  (  interface_dcl     forward_dcl  )

interfaceContent  :
  type_dcl  ;     const_dcl  ;     except_dcl  ;     attr_dcl  ;
  op_dcl  ;     keyStatement  ;

interface_body  :
  (  {  properties  }  interfaceContent  )*

interface_dcl  :
  simple_declarator  {  inheritance_spec  }  {  interface_body  }

javaAccessMode  :
  public     protected  |  private  |  packageprivate  | none

keyStatement  :
  key  simple_declarator  attributeList

literal  :
  integer_literal     floating_point_literal     boolean_literal
  string_literal    character_literal

lockMode  :
  normal     dirtyread

member  :
```

```
  type_spec  declarators  ;

member_list  :
  (  member  )+

module  :
  module  simple_declarator  {  (  {  properties  }  definition  )+  }

native_type  :
  native  (  scoped_name  (  *  )*  )* simple_declarator

object_type  :
  Object

octet_type  :
  octet

op_attribute  :
  oneway

op_dcl  :
  {  op_attribute  }  op_type_spec  simple_declarator  parameter_dcls
    {  raises_expr  } {  context_expr  }

op_type_spec  :
  param_type_spec     void

outerStateSetStatement  :
  stateSetSpec  =  initState

packageBlock  :
  {  (  {  properties  }  definition  )*  }

packageScope  :
  simple_declarator  ::

packageStatement  :
  package  simple_declarator  packageBlock  ;

param_attribute  :
  in     out     inout

param_dcl  :
  param_attribute  param_type_spec  simple_declarator

param_type_spec  :
  base_type_spec     string_type     wide_string_type     fixed_pt_type
  scoped_name

parameter_dcls  :
  (  {  param_dcl  (  ,  param_dcl  )*  }  )

positive_int_const  :
  integer_literal

pragmaStatement  :
  pragma  include filename

properties  :
  [  property  (  ,  property  )*  ]

property  :
  assignedProperty     booleanProperty

qualifiedName  :
  simple_declarator  (  .  simple_declarator  )*
```

```
quoted_identifier  :
  identifier     string-literal

raises_expr  :
  raises  (  scoped_name  (  ,  scoped_name  )*  )

relationshipStatement  :
  relationship  simple_declarator
  {  roleStatement  ;  (  roleTriggerStatement  ;  )*
    {  roleStatement  ;  (  roleTriggerStatement  ;  )*
       {  assocSpec  ;  (  roleTriggerStatement  ;  )*  }
     }
  }

roleMult  :
  1..1     0..1     1..*     0..*

roleStatement  :
  {  annotations  }  role  scoped_name simple_declarator
                   roleMult  scoped_name { keyStatement  ; }

roleTriggerStatement  :
  trigger  scoped_name  upon  (  relate     unrelate  )
           simple_declarator

scoped_name  :
  {  ::  }  simple_declarator  (  ::  simple_declarator  )*

sequence_type  :
  sequence  <  simple_type_spec  {  ,  template_bounds  }  >

signal_dcl  :
  signal  simple_declarator  parameter_dcls

signed_int  :
  signed_long_int     signed_short_int     signed_longlong_int

signed_long_int  :
  long

signed_longlong_int  :
  long  long

signed_short_int  :
  short

simple_declarator  :
  quoted_identifier

simple_type_spec  :
  base_type_spec     template_type_spec     scoped_name

socleContent  :
  packageStatement     actionStatement     cppStatement

socleStatement  :
  (  {  properties  }  socleContent  )+

stateName  :
  simple_declarator

stateSetSpec  :
  stateset  {  stateName  (  ,  stateName  )*  }

string_literal  :
  string-literal
```

```
string_type  :
  string  {  <  template_bounds  >  }

struct_type  :
  struct  simple_declarator  {  inheritance_spec  } {  member_list  }

swalBlock  :
  {  delimited-action-language  }

switch_body  :
  (  case_clause  )+

switch_type_spec  :
  integer_type     char_type     boolean_type     enum_type     scoped_name

template_bounds  :
  positive_int_const     scoped_name

template_type_spec  :
  sequence_type     string_type     wide_string_type     fixed_pt_type

transitionStatement  :
  transition  qualifiedName  to  transitionTarget
                              upon  simple_declarator

transitionTarget  :
  qualifiedName     cannothappen     ignore

type_dcl  :
  typedef  type_declarator     struct_type     union_type     enum_type
            native_type

type_declarator  :
  type_spec  declarators

type_spec  :
  simple_type_spec     constr_type_spec

union_type  :
  union  simple_declarator  switch  (  switch_type_spec  )
        {  switch_body  }

unquoted_scoped_name  :
  {  ::  }  identifier  (  ::  identifier  )*

unsigned_int  :
  unsigned  (  unsigned_short_int     unsigned_long_int
                unsigned_longlong_int  )

unsigned_long_int  :
  long

unsigned_longlong_int  :
  long  long

unsigned_short_int  :
  short

wide_char_type  :
  wchar

wide_string_type  :
  wstring  {  <  template_bounds  >  }
```

# 4

# Action language reference

This chapter provides a reference to the Kabira Infrastructure Server (KIS) action language, organized by keyword. Each statement or function in the action language appears on its own page, with a top-level syntax definition, a description of the statement or function, and an example. See the section called “Complete action language grammar” on page 99 for a complete syntactic definition of the action language.

## About the notation

The syntax description for each statement or function uses the following conventions:

**bold** screen font indicates a literal item that appears exactly as shown

***italics*** indicates a language element that is defined elsewhere

`plain text` indicates a feature of the notation, as follows:

( item ) parentheses group items together

( item )* the item in parentheses appears zero or more times

( item )+ the item in parentheses appears one or more times

{ item } items in braces are optional

( item1 | item2 ) the vertical bar indicates either item1 or item2 appears

For example:

```
keyword { optionalKeyword }
     { optionalThing }( zeroOrMoreRepeatingThing )*
     ( oneAlternative | otherAlternative | thirdAlternative )
```

# Some common elements

The following paragraphs describe some common elements found in the action language grammar. These are not statements or functions, so they do not appear under their own headings later in this chapter. Nor does the action language grammar (see the section called “Complete action language grammar” on page 99) explain their meaning.

## handle

A handle is a reference to an object. When initially declared, a handle is empty and may be used only as the target of an assignment, select, or create statement.

## chainSpec

A chainSpec represents a relationship traversal. When one entity or interface has a relationship to another entity or interface, two objects may be related using a relate statement. The relationship can then be traversed from one object to the other; this involves a chain spec of the form:

scopedType [ roleName ]

For example, the for..in statement:

for handle in setSpec { where whereSpec } statementBlock

accepts either a chainSpec or an entity name for setSpec. The chainSpec might be used as in either of the following examples:

```
// verify this subscriber's phone services
for s in thisSubscriber->Service[subscribesTo]
{
    service.verify();
}

// reorder all the parts used in this assembly
for p in thisAssembly->Component[contains]->Part[includes]
{
    part.reorder();
}
```

A chainSpec may also represent an associative relationship traversal. An associative relationship is where one entity has a relationship with another. This relationship can only be traversed using the third entity. For example:

```
select thisSeller from Seller where thisSeller.name = "fred";

for aBuyer in thisSeller->Sale[sell_to]->Buyer[sell_to]

{

   aBuyer.verifyCredit();

}
```

# assert

Model level assertion.

## Syntax

```
assert ( expression );
```

## Description

expression can be any valid boolean expression defined in IDLos.

When an assertion fails, the runtime logs an error message as a fatal error in the trace file and sends the error message to `stderr` . The message includes the model file name and line number where the assertion failed.

The following is an example of an assertion message:

```
x.soc:59: failed assertion '(idx > 1)'
```

After the runtime logs the message, it throws a runtime exception in the model.

## IDLos constraints

None.

## Exceptions

None.

## Warnings

Assertions are enabled only for DEVELOPMENT builds; they do not appear in PRODUCTION builds.

## Examples

```
assert( !empty myobj );
assert( idx >= 1 );
assert( myEnum == One    myEnum == Two );
assert( self.boolOp() == true );
```

## See also

# break

Exit from a while or for loop.

## Syntax

```
break;
```

## Description

This loop control statement lets you exit immediately from a while, for...in, or for loop. If while or for loops are nested, the break exits from the containing loop only.

## IDLos constraints

None.

## Exceptions

None.

## Warnings

None.

## Example

```
declare long x;
for (x = 0; x < 10; x++)
{
    //
    //    Check for termination condition.  If
    //    found get out of loop immediately
    //
    if (someCondition)
    {
        break;
    }
}
```

## See also

the section called " continue" on page 66 , the section called " for" on page 76 , the section called " for ... in" on page 77 , the section called "return" on page 83 , the section called "while" on page 98

# cardinality

Return the number of objects in an extent or relationship.

## Syntax

```
cardinality ( ( className   (handle   self) -> chainSpec ) );
```

## Description

cardinality returns the number of objects in an extent or relationship. Cardinality can be used on extents and a single relationship chain. Extended relationship chains are not allowed.

cardinality should be used instead of iterating over an extent or relationship to determine the number of objects. This has a large performance advantage over iteration since extents and relationships maintain the number of objects directly, eliminating the requirement to iterate over the entire extent or relationship.

## IDLos constraints

None

## Exceptions

None.

## Warnings

When you use cardinality on extentless entities, the result is approximate.

## Example

```
entity E { };
entity I { };

relationship EToI
{
    role E toI 1..1 I;
    role I toE 1..* E;
};

// --- Get the number of instances of E (the extent) ---
declare long numE = cardinality (E);

// --- Get the number of instances of E related to I ---
declare long numIToE = 0;
declare I    anI;

// Iterate over the extent adding up the number of
// related instances of E
for anI in I
{
    numIToE = numIToE + cardinality (anI->E[toE]);
}
```

## See also

the section called “ for ... in” on page 77 , the section called “in” on page 78 ,

# Conditional operator "?"

Conditionally execute a statement.

## Syntax

```
relationalExpression
(
        booleanOperator relationalExpression
)*
{ "?" expression ":" expression }
```

## Description

A simple `if .. else` instance can be replaced by the *conditional operator*. For example:

```
result = (var1 == var2) ? trueStatement : falseStatement;
```

If the expression represented by (var1 == var2) is true, result will be assigned the value of trueStatement, otherwise the value of falseStatement. The conditional operator can be used wherever an expression can be used.

## Exceptions

None.

## Warnings

None.

## See also

the section called “if else else if” on page 79

# continue

Skip immediately to the next iteration of a for, for...in, or while statement.

## Syntax

```
continue;
```

## Description

This loop control statement skips immediately to the next iteration of a for, for...in, or while statement.

## IDLos constraints

None.

## Exceptions

None.

## Warnings

None.

## Example

```
declare Customers aCustomer;
for aCustomer in Customers
{
    //    Check to see if we have already processed
    //    this customer.  If we have, skip this customer
    //    and proceed to the next

    if (aCustomer.seen)
    {
        continue;
    }

    //    First time we have seen this customer.  Welcome
    //    them.
}
```

## See also

the section called " break" on page 63 , the section called " for" on page 76 , the section called " for ... in" on page 77 , the section called "return" on page 83 , the section called "while" on page 98

# create

Create an object with initial values.

## Syntax

```
create handle { values ( valueSpec (, valueSpec )* ) } ;
```

## Description

This statement creates an object using a previously declared handle of entity or interface type. A handle is set to empty upon declaration, and is invalid (that is, it does not refer to any object) until it is assigned to a valid object or used in a create statement.

valueSpec is an initial value assignment for the attributes or relationships, in the form attribute-name:value (for relationships, chainSpec:value) If attributes are not assigned initial values, the value of the attribute is undefined. The only exception to this is object references. Object references are initialized to a value of empty.

## IDLos constraints

The entity being created cannot have been declared as a singleton.

# Exceptions

```
swbuiltin::ExceptionObjectNotUnique
```

A duplicate key was detected. Update the values statement to contain unique values for all keys in the object.

```
swbuiltin::ExceptionResourceUnavailable
```

A resource required to create the object was unavailable. This can happen if this is a create of a remote or a persistent object.

# Warnings

It is an error to use create on entities that have been declared as singleton in your model. You must use the section called “ create singleton” on page 69 instead.

Attributes that are part of a key must be initialized to a unique key value to avoid duplicate key exceptions during object creation.

Relationships that are used for referential integrity in a backing database should be initialized during creation to ensure that referential integrity is maintained.

# Examples

```
entity Customer
{
    attribute string   name;
    attribute long     id;
    key CustId { id };
};

declare Customer  lclCustomer;
declare Customer  rmtCustomer;
declare long      uniqueKey;

//
// allocateCustId allocates a unique customer id
//
uniqueKey = self.allocateCustId();

//
// Create a customer on the local node
//
create lclCustomer values (name:”Joe”, id:uniqueKey);

//
// Allocate another customer id
//
uniqueKey = self.allocateCustId();

//
//    Create a customer
//
create rmt Customer values (name:”Jean-Louis”, id:uniqueKey);
```

# See also

the section called “ create singleton” on page 69 , the section called “ delete” on page 71 ,

# create singleton

Create a singleton object with initial values.

## Syntax

```
create singleton  handle { values ( valueSpec (, valueSpec )* ) };
```

## Description

This statement creates a singleton object using a previously declared handle of entity or interface type. There is only one instance of a singleton entity.

create singleton can be called multiple times. If the single instance has not been created it is created and returned to the caller. If the instance already exists, a handle to the single instance is returned to the caller.

The initial values specified are only applied if the singleton is created. If the singleton was created by a previous create singleton, the initial values have no affect.

## IDLos constraints

Entity must be declared as a singleton.

## Exceptions

```
swbuiltin::ExceptionResourceUnavailable
```

A resource required to create the object was unavailable. This can happen if this is a create of a remote singleton.

## Warnings

It is an error to use create singleton on entities and interfaces that have not been declared as a singleton in IDLos.

Initial values are only applied when the singleton is created.

## Example

```
[ singleton ]
entity PortAllocator
{
    void doInitialization();
};

//
// Create a singleton
//
declare PortAllocator myAllocator;

create singleton myAllocator;

//
//        Invoke a two-way operation to ensure that
```

```
//        it is initialized before using it
//
myAllocator.doInitialization();
```

## See also

the section called “ create” on page 67 , the section called “ delete” on page 71

# declare

Declare a variable.

## Syntax

```
declare type identifier { [ integerLiteral ]   assignmentExpression };
declare { ( type identifier { [ integerLiteral ]
                                                  assignmentExpression }; )+ }
declare const type identifier { [ integerLiteral ]   assignmentExpression };
```

## Description

All variables must be declared before they are used. The KIS action language is strongly typed. Variables are scoped to an operation, state, or code block defined within an operation or state.

The value of a variable is undefined until it is assigned a value. The only exception is object handle which are initialized to empty when they are declared. Variables can be assigned initial values in the declare statement.

Variables can be declared as arrays. There is no support for initializing the values of an array in the declare statement.

Strings can also be declared as either bounded or unbounded.

## IDLos constraints

None.

## Exceptions

None.

## Warnings

A variable should always be initialized to a known value. The value of an uninitialized variable is undefined and may change in the future. The only exception is for object handles, which are empty after being declared.

## Example

```
//
// Declare some variables
//
```

```
declare long      aLong = 0;
declare AnEntity  anEntity;

//
// Declare an array of shorts
//
declare short     anArrayOfShorts[100];

//
// Declare a variable inside of a for loop
//
declare long i;
for (i = 0; i < 10; i++)
{
    declare long tmpVar;
}
// tmpVar is out of scope here

//
// Bounded strings
//
declare string<10> firstName;

//
// Unbounded string
//
declare string  unboundedString;
```

## See also

the section called “ Types” on page 95

# delete

Destroy an object.

## Syntax

```
delete handle;
```

## Description

Destroy an object. All memory associated with the object is discarded. After invoking delete on an object handle, using that handle will cause a system exception to be thrown.

An implicit unrelate is performed on all relationships in which this object participates.

## IDLos constraints

None.

## Exceptions

```
swbuiltin::ExceptionResourceUnavailable
```

A resource required to create the object was unavailable. This can happen if this is a delete of a remote or a persistent object.

# Warnings

Be careful when deleting one of multiple references to an object: any outstanding references will be invalid once the object is deleted. However, within the same transaction as the delete the object reference remains valid, as shown in the following example. This example will print “Setting value” when the model is run:

```
package P
{
    entity E
    {
        attribute long i;
    };

    [ local ] entity StartUp
    {
        [ initialize ] void doIt();
    };

};

action P::StartUp::doIt
{`
    declare E e1;
    declare E e2;

    create e1;

    // Make both handles refer to same instance of E
    e2 = e1;

    // Instance will be deleted when transaction commits.
    // Meanwhile the reference is still valid
    delete e1;

    // e2 refers to an object scheduled for deletion,
    // but right now the reference is still valid
    if (!empty e2)
    {
        // This is reached because the reference is still valid
        printf("Setting value\n");
        e2.i = 1;
    }

    // Instance deleted on return (transaction commits)
`};
```

# Example

```
declare Customer aCustomer;
create aCustomer;

// Do something with customer

// Destroy the customer
delete aCustomer;

// aCustomer handle is now invalid
```

# See also

the section called “ create” on page 67 , the section called “ create singleton” on page 69

# empty

Test and set handles to empty.

## Syntax

```
declare type_identifier handle = empty;
empty  handle
handle = empty;
```

## Description

empty returns a boolean value if used in an expression. True is returned if the handle is currently unassigned to a valid object reference. False is returned if the handle holds a valid object reference.

empty can also be used to initialize a handle to known value that indicates that the handle does not contain a valid object reference.

## IDLos constraints

None

## Exceptions

None.

## Warnings

None

## Example

```
entity Customer { };

...

declare Customer aCustomer = empty;

//
// If the customer handle is empty create a new customer
//
if (empty aCustomer)
{     create aCustomer;
}

//
// Set the customer handle to empty
//
aCustomer = empty;
```

## See also

# Exceptions

Exception handling.

## Syntax

```
try statementBlock ( catchStatement )+
catch( scopedType ) statementBlock
throw handle;
```

## Description

These statements provide support for handling user and system exceptions.

User exceptions are defined in IDLos using the IDL exception and raises keywords. System exceptions are a predefined set of exceptions that can be raised by the KIS runtime and adapters. System exceptions cannot be thrown by the user, but they can be caught.

The supported system exceptions are defined in the swbuiltin component. You must import this component to catch system exceptions in action language. For details on the individual system exceptions, refer to the on-line documentation for the swbuiltin component.

## IDLos constraints

User exceptions are defined using IDLos. Operations that throw an exception must have a raises clause in the operation signature.

## Exceptions

None.

## Warnings

Uncaught user and system exceptions will cause the current oneway operation to be discarded or the engine will exit with an error depending on a configuration value.

System exceptions should not be thrown by users.

Handling the system exceptions ExceptionDeadLock and ExceptionObjectDestroyed in application code not executed as part of spawn will produce unpredictable but bad results.

## Example

```
package Life
{
   entity A
   {
       exception IsDead
       {
    string  causeOfDeath;
```

```
       };
       exception IsTired
       {
    long    nextTime;
       };
       void wakeUp(in string reason) raises (IsDead, IsTired);
       void checkIt(void);
    };
;};

//
//  wakeUp operation.  Throws exception on how
//  the wakeup was processed
//
action Life::A::wakeUp
{`
    //
    //    Figure out my current state
    //
    if (imDead)
    {
        declare IsDead          id;
 id.causeOfDeath = "boredom";
        throw id;
    }
    if (imTired)
    {
        declare IsTired         it;
 it.nextTime = 60;
        throw it;
    }
`};

//
//        checkIt handles exceptions thrown by wakeUp
//
action Life::A::checkIt
{`
    try
    {
        self.wakeUp("it's noon!");
 }
    catch (IsDead id)
    {
        declare string cd = id.causeOfDeath;
 printf("id.causeOfDeath = %s\n", cd.getCString());
    }
    catch (IsTired it)
    {
 printf("it.nextTime = %s\n", it.nextTime);
    }
`};

//
//   Example of string overflow exception
//
entity StringOverflow
{
    attribute string<5>  stringFive;
};

//
//   Assign a string larger than 5 characters to
//   stringFive.  This will cause an exception
//
try
{
```

```
    self.stringFive =”123456”;
}
catch (ExceptionStringOverflow)
{
    printf(“caught ExceptionStringOverflow\n”);
}

//
//   Example invalid array exception
//
typedef long ArrayFive[5];

//
//   Use an index > 4.  This will cause an exception.
//   Remember that arrays are zero based.
//
declare ArrayFive arrayFive;
try
{
    arrayFive[5] = 1;
}
catch (ExceptionArrayBounds)
{
    printf(“caught ExceptionArrayBounds\n”);
}
```

## See also

the section called “spawn” on page 88 , the section called “ Transactions” on page 93

# for

Iterate over statements with iterator expression and termination condition.

## Syntax

```
for ( lval { assignmentExpr } ; conditionExpr ; iteratorExpr )
      statementBlock
```

## Description

An iteration statement; enables you to loop through and execute a block of code until a condition becomes false.

assignmentExpr is performed once before iteration begins. It is typically used to set the initial value of a variable used in the next two expressions.

conditionExpr is evaluated before each iteration. If the expression is nonzero, then statementBlock is executed.

iteratorExpr is performed each iteration after statementBlock is executed.

## IDLos constraints

None.

## Exceptions

None.

## Warnings

None.

## Example

```
for ( x = 0; x < maxTimes; x++ )
{
    //
    //    Repeat this work maxTimes
    //
}
```

## See also

the section called “ for ... in” on page 77 , the section called “while” on page 98

# for ... in

Iterate over an extent or relationship.

## Syntax

```
for handle in setSpec
    { { using className } where whereSpec }
    { readlock | writelock | nolock }
    statementBlock;
```

## Description

Iterates over a set of objects. If the set is empty, the statement block is never executed. The handle must be of the same type as the set it iterates over.

## IDLos constraints

None.

## Exceptions

```
swbuiltin::ExceptionStringOverflow
```

The length of a bounded string attribute was exceeded in a value specified for that attribute in the where clause.

## Warnings

None.

## Example

Consider the following IDLos definitions:

```
entity Customer { attribute string firstName; };
entity Queues { };

relationship CustomerQueues
{
    role Queue holds 0..* Customer;
    role Customer waitsIn 1..1 Queue;
};
```

Queues and Customers that have been created and related can be iterated across using the for...in statement in various ways:

```
//
// --- welcome all the customers held in some queue q ---
//
for cust in q->Customer[holds]
{
    //    Welcome this customer
}

//
// --- Send a bill to every customer that exists ---
//
for cust in Customer
{
    //    Send a bill to this customer
}

//
// --- Welcome customers called Dan in some queue q ---
//
for cust in q->Customer[holds] where (cust.firstName == “Dan”)
{
    //    Welcome this Dan
}
```

## See also

the section called “cardinality” on page 64 , the section called “select” on page 84 , the section called “in” on page 78

# in

Test whether a given object reference refers to a member of some set of objects.

## Syntax

```
handle in handleOrSelf  ->  chainSpec
```

## Description

Returns `true` if the object referenced by the handle is a member of the set specified. Otherwise it returns `false`.

## IDLos constraints

None.

## Exceptions

None.

## Warnings

Do not confuse this function with the for..in statement.

## Example

```
//
// If is one of my cars, drive it
//
if( thisCar in self->Car[owns] )
{
    thisCar.drive();
}
```

## See also

the section called “cardinality” on page 64 , the section called “ for ... in” on page 77

# if else else if

Conditionally execute a statement block.

## Syntax

```
if booleanExpression
    statementBlock
  { ( else
           statementBlock   elseIfStatement ) }

elseIfStatement :
else if booleanExpression
    statementBlock
  { ( else
      statementBlock   elseIfStatement ) }
```

## Description

A conditional control flow operator. The else clause is optional. If statements may be nested.

## IDLos constraints

None.

## Exceptions

None.

## Warnings

None.

## Example

```
if (aCustomer.balanceDue < aCustomer.creditLimit)
{
    //
    //    Allow withdrawal
    //
}
else if (aCustomer.isPaymentOverdue)
{
    //
    //    Refuse order with rude message
    //
}
else
{
    //
    //    Credit limit is expired, but payments
    //    current.  Politely refuse withdrawal
    //
}
```

## See also

the section called “Conditional operator "?"” on page 65 , the section called “ for” on page 76 , the section called “while” on page 98

# isnull

Test and set the null indicator of an attribute or of a struct or exception member.

## Syntax

```
isnull attributeName
attributeName = isnull;

isnull structInstance.memberName
structInstance.memberName = isnull;

isnull anException.memberName
anException.memberName = isnull;
```

## Description

isnull returns a boolean value if used in an expression. `true` is returned if the attribute or structure/exception member has never been assigned a value. `false` is returned in the attribute or

structure/exception member has been explicitly set. For object attributes, values provided at create time as well as initial values defined in the entity will also clear the null indicator.

isnull can also be used to set the null indicator of an attribute or structure/exception member.

## IDLos constraints

None.

## Exceptions

None.

## Warnings

If isnull is used to set the null indicator of an attribute or structure/exception member the data for the attribute or member does not change though subsequent tests for isnull will be `true`.

## Example

```
struct Item
{
 string name;
}
entity Invoice
{
   ...
    attribute long total;
    attribute Item item;
}
declare Invoice anInvoice;
create anInvoice;
...
declare Item item;
if (isnull anInvoice.total)
{
     // the attribute total is null.
}

item = anInvoice.item;
assert(isnull item.name);
```

# relate

Relates one object to another across a relationship.

## Syntax

```
relate (handle   self) relationshipSpec (handle   self)
        { using  (handle   self) } ;
```

# Description

Establishes a relationship between two objects, or an associative relationship between three objects if a using clause is provided.

Relationships can be used as ordered lists by defining a 1..1 relationship from an entity to itself. The order of the handles in the relate statement defines the order of the objects in the relationship.

The order of the two handles is significant only if they refer to the same entity.

If the cardinality of any side of a role is 1, performing a relate on that role will cause an implicit unrelate to occur for any objects previously in that relationship.

# IDLos constraints

None.

# Exceptions

None.

# Warnings

None.

# Example

```
entity Customer { };
entity Queues { };

relationship CustomerQueues
{
    role Queue holds 0..* Customer;
    role Customer waitsIn 1..1 Queue;
};

relationship CustomerOrder
{
    role Customer inFrontOf 1..1 Customer;
    role Customer inBackOf 1..1 Customer;
};

//
//    Put a customer in a queue
//
relate aCustomer waitsIn aQueue;

//
//    This is also valid for putting a customer
//    in a queue
//
relate aQueue holds aCustomer;

//
//    Put impatient customer in front of nice customer
//
relate impatientCustomer inFrontOf niceCustomer;
```

## See also

the section called “ unrelate” on page 97

# return

Return from an operation or state.

## Syntax

```
return { expression } ;
```

## Description

A return statement signals completion of an operation or a state and returns control to the caller. Operations with return values must have a return statement and provide a return value expression.

There may be any number of return statements in an operation or state.

## IDLos constraints

Operations must be declared with the correct return type.

## Exceptions

None.

## Warnings

It is illegal to return a value from a state.

## Example

```
entity ATM
{
    long getCash();
    void pressOk();
};

//
//    Implementation of getCash operation
//
action getCash
{‘
    declare long        cashAmount = 12345;
    return cashAmount;
‘};

//
//    Implementation of pressOk operation
//
action pressOk
{‘
    return;
‘};
```

## See also

# select

Select an object using a relationship or an extent.

## Syntax

```
select handle from (handle | self)
    ( ( -> ) chainSpec )+
    { { using className } where whereSpec }
    { readlock | writelock | nolock };

select handle from className
    { using className } where whereSpec
    { readlock | writelock | nolock };

select handle from singleton className;
```

## Description

select always returns a single object. It can be used against an extent or a many relationship only with a where clause to narrow the select to return a single object.

The three forms of select above are:

- navigate a relationship chain with a where clause. The where clause is not required if this navigation is towards the one side of a relationship role.
- select a single object from an extent using a where clause.
- select a singleton.

## Constraints

None.

## Exceptions

```
swbuiltin::ExceptionStringOverflow
```

The length of a bounded string attribute was exceeded in a value specified for that attribute in the where clause.

## Warnings

It is a build audit if a select statement returns more than a single object.

## Example

```
[ singleton ]
entity TheBoss { };
```

```
entity Customer
{
    attribute long  custId;
    key CustomerId { custId };
};
entity Contact { };

relationship CustomerContacts
{
    role Contact isFrom 1..1 Customer
    {
        key Id {custId };
    };
    role Customer has 0..* Contact;
};

///// ---- above was IDLos, below is action language ---- /////

//
//    Select the next customer in line
//
select aCustomer from aContact->Customer[isFrom];

//
//    Select next customer by customer identifier from a specific contact.
//    Force a writelock
//
select aCustomer from aContact->Customer[CustomerContacts::isFrom]
    using CustomerContacts::isFrom::Id where aCustomer.custId == 123 writelock.

//
//    Select customer 123 from all customers
//
select aCustomer from Customer where (aCustomer.custId == 123);

//
//    Find the boss
//
select boss from singleton TheBoss;
```

## See also

the section called “  for ... in” on page 77

# select...using

Select an object using a key.

## Syntax

```
select handle from className using keyName where whereSpec
          { readlock | writelock | nolock }
          { on empty create { values valuesSpec } }

select handle from singleton className using keyName where whereSpec
          { readlock | writelock | nolock }
          { on empty create { values valuesSpec } }
```

# Description

select...using looks up an entity using the specified key. By default, if the object is found, it is locked for reading. If you specify nolock, the object will not be locked. Specifying writelock locks the object for writing.

If you specify on empty create and the object is not found, select...using creates a new object with the key values you supplied in the where clause. You can set other attribute values using the optional values clause.

# Warnings

If no object is found, and on empty create was not specified, it is possible that an entity matching the specification may be created in another transaction. This could cause a subsequent create of the object to throw ExceptionObjectNotUnique.

If is a build audit failure if select can return more than one instance.

# Exceptions

```
swbuiltin::ExceptionStringOverflow
```

The length of a bounded string attribute was exceeded in a value specified for that attribute in the where clause.

# Examples

The following examples are based on this IDLos entity definition:

```
entity Obj
{
    attribute long x;
    attribute long y;

    key key1 { x };
};
```

To select the instance of entity Obj where x is zero:

```
declare Obj obj;
select obj from Obj using key1 where (obj.x == 0);
```

If no instance exists where x is zero, the handle "obj" will be empty. To create an object with the given key if it is not found, use on empty create:

```
select obj from Obj using key1 where (obj.x == 0)
on empty create;
```

This instantiates an object with the attribute x initialized to zero. The attribute y will be unset. To initialize additional attributes when the entity is created, a values clause may be used:

```
select obj from Obj using key1 where (obj.x == 0)
on empty create values (y:46);
```

Note that the key attributes' values are taken from the where clause, and may not be specified again in the values clause.

## Locking examples

By default, when an entity is found, it is locked for reading. This ensures that the entity will not be modified by another transaction after it is selected. If the entity is to be modified, a write lock may be taken via the writelock option:

```
select obj from Obj using key1 where (obj.x == 0) writelock;
```

This prevents a lock promotion if a write lock is needed later, eliminating the possibility of a lock promotion deadlock.

The nolock option may be used to specify that the object should not be locked. This is especially useful when sending oneway events to objects which may be locked by another transaction:

```
select obj from Obj using key1 where (obj.x == 0) nolock;
```

Using the nolock option means that another transaction can modify the object you have just selected, while you are handling it.

# self

Reference the current object.

## Syntax

```
self
```

## Description

The self keyword is used to reference the current object in which an operation or state is executing.

In spawned operations, self refers to the thread of execution, not to an actual object. See the section called “spawn” on page 88 for more information.

## IDLos constraints

None.

## Exceptions

None.

## Warnings

Assigning empty or any handle to self is illegal.

## Example

```
entity Customer
{
    attribute long  customerId;
    void anOperation();
```

```
};

action anOperation
{‘
    declare long x;
    x = self.customerId; // Access customerId in the current object
‘};
```

## See also

the section called “ empty” on page 73

# spawn

Spawn a user thread of control.

## Syntax

```
spawn scopedType ( { paramSpec ( , paramSpec )* } );
```

## Description

Spawn is used to create a thread of control that is managed by the user. This provides a mechanism for users to call blocking external functions and manage transactions as work is injected into KIS.

For example, spawn can be used to create a listener thread for a protocol stack. As messages are received on the protocol stack a transaction is started using the transaction support in action language and the work is injected into KIS.

The spawn statement can only be used on operations in local entities. Thus the spawned operation is not associated with any transaction or shared memory.

Object references used as parameters to the spawn operation and ones declared in the spawn operation are not in a transaction until a begin transaction statement is executed. Access to these objects remains valid until a commit or abort transaction statement is executed. Accessing these objects outside of a transaction will cause a runtime exception.

For example, the following action language will cause a runtime exception:

```
ALocalEntity::aSpawnMethod(in long arg1)
{
    declare AnEntity    anEntity;

    // Causes a runtime exception since no transaction
    //    was started
    create anEntity;

    begin transaction;
        ...
}
```

## IDLos constraints

You can only spawn an operation that:

- is defined in a local entity

- is a two-way void operation
- uses only in parameters

# Exceptions

None.

# Warnings

The spawn verb in action language is non-transactional. This implies that if the spawn verb is called in an operation that is retried because of deadlock, the spawn of the original thread will already have happened and will not be rolled back. Modelers must take caution to ensure that operations that are using spawn do not get retried.

One way to avoid retries is to invoke spawn from a commit trigger.

Spawned operations must use begin transaction before accessing any objects. Similarly, they must use commit transaction once they have finished.

Spawned operations must explicitly manage all exception handling, including transaction deadlocks. (A transaction deadlock can occur when accessing an object reference, accessing an attribute, or invoking a two-way operation.) The spawned operation must catch the ExceptionDeadLock system exception and abort the current transaction. Any uncaught system exceptions will cause a fatal engine failure.

Spawned operations cannot use `return` while in an active transaction. Doing so causes a fatal engine error.

When an engine is shut down, the thread manager kills spawned threads that are not in an active transaction. During transactions in spawned threads, you should periodically check whether the thread needs to terminate. Refer to the online documentation for the `swbuiltin` component to learn more about using the `isStopping()` operation.

A spawned thread that blocks in an external function with a transaction active can prevent clean shutdown. Do not block within transactions in spawned threads.

# Example

The following example shows how the `spawn` and `transaction` statements can be used to implement a simple server.

The example illustrates the use of an external library to access some resource (for example, read from a file, listen on a socket, etc.). The example uses a made-up external library `somelib` that provides access via the operations `createConnection()` and `getMessage()`. The library and operations are not implemented, but are presented here for illustration only.

```
package AServer
{
    entity Server
    {
        void onCreate();
        void onCommit();

        trigger onCreate upon create;
```

```
        trigger onCommit upon commit;
    };

    [ local ]
    entity ThreadWrapper
    {
        void aThread(in string workerName);
    };

    // Worker implements the transactional
    // work within KIS
    //
    entity Worker
    {
        attribute string name;

        key primary {name};

        oneway void processMessage(in string message);
    };
};

// Implementation of Server
// To start the server, the application creates
// an instance of Server.
//
void
action AServer::Server::onCreate()
{`
    declare swbuiltin::ObjectServices os;
    declare Worker worker;

    // Create a worker for injecting work into the application.
    //
    create worker values (name: "aWorker");

    // Enable the commit trigger for this transaction.
    // When the  creation commits, the commit
    // trigger will start the server
    //
    os.enableCommitTrigger(self);
`};

void
action AServer::Server::onCommit()
{`
    // Initialize (e.g. establish a connection to)
    // the external resource
    //
    connection = somelib.createConnection();

    // Stash the handle of the connection in the
    // worker object
    //
    ...

    // Spawn a thread of control for the worker.
    //
    spawn ThreadWrapper::aThread("aWorker");
`};

// Implementation of aThread
//
void
action AServer::ThreadWrapper::aThread(
    in string workerName)
{`
```

```
    declare swbuiltin::EngineServices engineServices;
    declare string message;
    declare boolean haveMessage = false;
    declare Worker worker;

    // To allow clean shutdown, periodically
    // check whether the thread needs to terminate.
    //
    while (!engineServices.isStopping())
    {
        if (!haveMessage)
        {
            // Get a message from a nontransactional source.
            // This call to an external library is not within
            // a transaction, so it is OK if this call blocks.
            //
            message = somelib.getMessage();
            haveMessage = true;
        }

        // Process the message in KIS within a transaction
        //
        begin transaction;

        // Catch retryable exceptions, including deadlocks
        // and object destroyed exceptions.
        // All other exceptions will bubble up and cause
        // the engine to fail.
        //
        try
        {
            select worker from Worker using primary
                where (worker.name == workerName);

            assert( !empty worker );

            worker.processMessage(message);
        }
        catch (swbuiltin::ExceptionRetryable)
        {
            // Upon deadlock or other retryable exception,
            // abort the tranaction and try again using
            // the same message data.
            //
            abort transaction;
            continue;
        }

        // Successfully processed the message, commit and
        // get more data
        //
        commit transaction;
        haveMessage = false;
    }
`};
```

## See also

the section called “ Exceptions” on page 74, the section called “ Transactions” on page 93

# super

Access a supertype implementation of a virtual operation.

# Syntax

```
super.operation();
```

# Description

The `super` keyword is used to invoke a supertype implementation of an operation on the current object which is `virtual` on a supertype of the current object.

# IDLos constraints

The `super` keyword can only be used to invoke virtual operations. It is not an object reference, therefore you cannot access attributes or assign empty or any handle to `super`.

# Exceptions

None.

# Warnings

`super` is not an object reference. All of the following statements will fail to parse.

```
    super.id = 1;          // illegal
    someValue = super.id;  // illegal
    delete super;          // illegal
    super = empty;         // illegal
    super = someObject;    // illegal
```

# Example

```
entity a
{
    [ virtual ]
    void op1();

    [ virtual ]
    void op2();
};

entity b : a
{
    [ virtual ]
    void op1();
};

...

action b:op1()
{`
    // invokes a::op1()
    super.op1();

    // invokes a::op2()
    super.op2();
`};
```

## See also

the section called “self” on page 87

# test_assert

Model level assertion.

## Syntax

```
test_assert ( expression );
```

## Description

expression can be any valid boolean expression defined in IDLos.

When an assertion fails, the runtime logs an error message as a fatal error in the trace file and sends the error message to `stderr` . The message includes the model file name and line number where the assertion failed.

The following is an example of an assertion message:

```
x.soc:59: failed assertion '(idx > 1)'
```

After the runtime logs the message, it throws a runtime exception in the model.

## IDLos constraints

None.

## Exceptions

None.

## Warnings

Test assertions are always enabled.

## Examples

```
test_assert( !empty myobj );
test_assert( idx >= 1 );
test_assert( myEnum == One    myEnum == Two );
test_assert( self.boolOp() == true );
```

# Transactions

Explicitly manage KIS transactions.

# Syntax

```
( begin   commit   abort ) transaction ;
```

# Description

The three types of transaction statements let you explicitly manage transactions. This is only required and only allowed in an operation that was invoked using the spawn statement. All other operations and states are implicitly in a transaction when they are executed.

Objects that are created or selected in a spawned operation must be done in the context of a valid transaction that was started using begin transaction.

Once an object has been created or selected in a valid transaction it is implicitly reattached to any new transactions

See also “Transaction processing” on page 243 and related topics for a discussion of how transactions, locks, and deadlocks are handled at run time.

# IDLos constraints

None.

# Exceptions

```
swbuiltin::ExceptionDeadLock
```

A deadlock was detected. The current transaction should be aborted and retried.

```
swbuiltin::ExceptionObjectDestroyed.
```

An object accessed in the current transaction is being destroyed in another transaction. The current transaction should be aborted and retried.

# Warnings

Accessing objects passed as parameters are declared in a spawn operation that are not in a valid transaction will cause a runtime exception.

# Example

This is a simple example of creating and selecting objects in a spawned local operation. A more detailed example of using transaction control is also provided under the section called “spawn” on page 88.

```
ALocalEntity::aSpawnOperation(in long arg1)
{
    declare MyObject     mobj;
    declare SelectObject sobj;

    //
    //    Create and select in a valid transaction
    //
    begin transaction;
    create mobj;
```

```
    select sobj from singleton SelectObject;
    commit transaction;

    for (;;)
    {
        begin transaction;

        //
        //    No need to select these objects again.
        //    They are implicitly reattached to the new
        //    transaction.
        //
        mobj.numThings++;
        sobj.doIt(mobj.numThings);

        commit transaction;
    }
}
```

## See also

the section called " Exceptions" on page 74, the section called "spawn" on page 88

# Types

Action language data types.

## Syntax

See IDLos chapter.

## Description

The action language uses the IDL type system, just like the rest of IDLos does. The declare statement is used to define a type in an operation or state.

Structure members are accessed using the "." notation.

Arrays and sequences are both accessed using "[n]" indexing to access a specific element in an array or sequence. Arrays and sequences use zero based indexing. Sequence length uses an unsigned long long value. You must use long long or unsigned long long types when iteratoring over a sequence, otherwise a compilation warning may be generated.

Unbounded sequences are automatically grown at runtime as required. If the index being accessed is larger than the current sequence size, the sequence is resized. When a sequence is resized, the values of all new sequence members is undefined. They must be initialized to a known value by the application.

## IDLos constraints

None.

## Exceptions

```
swbuiltin::ExceptionArrayBounds
```

An array bounds error was detected by the runtime.

```
swbuiltin::ExceptionStringOverflow.
```

A string overflow was detected by the runtime. This can happen when the size of a bounded string is exceeded.

# Warnings

None.

# Example

```
struct AStruct
{
    long    aLongMember;
    short    aShortMember;
};

typedef sequence<char> UnboundedCharSequence;
typedef long ALongArray[10];

..... Above was IDLos, below is action language .....


// --- Accessing a structure member ---
declare AStruct    aStruct;
declare long       aLong;
aLong = aStruct.aLongMember;

// --- Accessing index 23 in unbounded sequence.  ---
// --- NOTE - Sequences and arrays are zero based. ---
declare UnboundedCharSequence   seq;
declare char                    aChar;
aChar = seq[22];

// --- Accessing the first index in an array ---
declare ALongArray   array;
declare long         aLong;
aLong = array[0];

// --- Using an any ---
declare any    anAny;
declare long   aLong;
declare string aString;

// --- Assign a long to an any ---
aLong = 1;
anAny <<= aLong;
// anAny now contains 1

// --- Now assign a string to the same any ---
aString = “hello world”;
anAny <<= aString;
// anAny now contains “hello world”
```

# See also

the section called “declare” on page 70 , the section called “ Exceptions” on page 74

# unrelate

Unrelate objects.

## Syntax

```
unrelate  (handle   self) relationshipSpec  (handle   self)
          { using  (handle   self) } ;
```

## Description

The unrelate statement terminates the relationship between the specified objects. If the relationship is an associative relationship the using clause must be specified to define the third-party in the associative relationship. Deleting a related object performs an implicit unrelate.

## IDLos constraints

None.

## Exceptions

None.

## Warnings

None.

## Example

```
entity Customer { };
entity Queues { };

relationship CustomerQueues
{
    role Queue holds 0..* Customer;
    role Customer waitsIn 1..1 Queue;
};

..... Above was IDLos, below is action language .....

// --- Remove a customer from a queue ---
unrelate aCustomer waitsIn aQueue;

// --- Remove a customer from a queue the other way works too ---
unrelate aQueue holds aCustomer;
```

## See also

the section called “ delete” on page 71 , the section called “relate” on page 81

# while

Loop over a statement block while a condition remains true.

## Syntax

```
while booleanExpression statementBlock
```

## Description

The while statement loops and executes a set of statements repeatedly as long as a condition is true. You may use break to exit a while loop at any time, or use continue to skip immediately to the next iteration.

## IDLos constraints

None.

## Exceptions

None.

## Warnings

None.

## Example

```
while (aClerk.atWork)
{
    // --- Can this clerk help this customer? ---
    if (!aCustomer.canHelp)
    {
        continue;
    }

    // --- Time to quit?  Just leave ---
    if (aClerk.doneForTheDay)
    {
        break;
    }

    //
    //    Service customers
    //
}
```

## See also

the section called “ break” on page 63 , the section called “ continue” on page 66 , the section called “ for” on page 76

# Complete action language grammar

This section contains the grammar that is used by the action language parser. Grammatical elements in this list occasionally differ from their equivalents in the action language statement and functions section, because in that section a few changes were made for the sake of clarity:

- minor name changes added missing semantic information
- some elements were expanded or condensed according to the grammar

The elements defined in the grammar are listed in alphabetical order. A few elements—such as string-literal or identifier—are defined by the IDL grammar. See the section called "Complete IDLos grammar" on page 54 for a definition of these items.

```
action :
 type scopedMethodName inputParameterList { const } statementBlock
 includeStatement   idStatement   package quotedIdentifier ;

actionFile :
 ( action )+

addingExpression :
 multiplyingExpression ( ( +   - ) multiplyingExpression )*

arrayExpression :
 { :: } ( quotedIdentifier :: )* quotedIdentifier   integerLiteral

arrayList :
 ( [ expression ] )*

assertStatement :
 ( assert booleanExpression | test_assert booleanExpression )

assignEqualOp :
 *=   /=   %=   +=   -=   <<=   >>=   &=
 ^=    =

assignmentExpression :
 ( =   assignEqualOp   shiftOp ) expression   ( ++   -- )

attributeName :
 quotedIdentifier

binaryOperator :
 +   -   *   /

bitExpression :
 signExpression ( ( &       ^ ) signExpression )*

booleanExpression :
 expression

booleanLiteral :
 TRUE   true   FALSE   false
booleanOperator :
 &&

cType :
 unsigned short   short   unsigned int   int
 long   long long   unsigned ( long
 long long )   float   double   unsigned char
 char   void
```

```
cardinalityFunction :
 cardinality ( scopedType   handleOrSelf -> chainSpec )

castExpression :
 castOperator < { const } ( scopedType   cType ) { * } >
 expression

castOperator :
 const_cast   dynamic_cast   reinterpret_cast
 static_cast

catchExpression :
 scopedType { quotedIdentifier }

catchStatement :
 catch catchExpression statementBlock

chainSpec :
 scopedType [ relationshipSpec ]

charLiteral :
 character-literal

className :
 scopedType

comparisonOperator :
 <   >   <=   >=   ==   !=

constDeclare :
 { const }

createStatement :
 create { singleton } handle { on locationSpec }
  { implementation implementationSpec }
  { values valueSpec ( , valueSpec )* }
declareSpecStatement :
 type quotedIdentifier { < integerLiteral > }
 {
  [ arrayExpression ] ;
  assignmentExpression ;
  ;
 }

declareStatement :
 declare declareSpecStatement
 declare { ( declareSpecStatement )+ }

deleteStatement :
 delete handle
 delete [ ] handle

elseIfStatement :
 else if booleanExpression statementBlock
 { ( elseStatement   elseIfStatement ) }

elseStatement :
 else statementBlock

emptyFunction :
 empty { handleOrSelf { . attributeName } { { expression } } }

expression :
 relationalExpression ( booleanOperator relationalExpression )*
 { "?" expression ":" expression }
```

```
extendedChain :
 ( ( -> ) chainSpec )+

externStatement :
 extern ( scopedType   cType ) quotedIdentifier

floatLiteral :
 floating-point-literal

forStatement :
 for handle in setSpec
  { using className }
  { where whereSpec }
  { order by attributeName }
  { readlock | writelock | nolock }
  statementBlock
 for lval { assignmentExpression } ; expression ; lval { assignmentExpression }
   statementBlock

genAssignDeclareStatement :
 { const } ( cType   scopedType   self )
  ( paramList
    userDereference quotedIdentifier
      { ( [ arrayExpression ]   assignmentExpression   paramList ) }
    . quotedIdentifier arrayList ( paramList
                              ( . quotedIdentifier )* = expression
                              assignEqualOp expression ) )

handle :
 quotedIdentifier

handleOrSelf :
 handle   self

idStatement :
 quotedIdentifier expression ;

ifStatement :
 if booleanExpression statementBlock { ( elseStatement   elseIfStatement ) }

implementationSpec :
 identifier   integerLiteral

inFunction :
 handle in handleOrSelf -> chainSpec

includeStatement :
 include filename

inputParameter :
 type & quotedIdentifier

inputParameterList :
 ( inputParameter ( , inputParameter )*   void   )

integerLiteral :
 decimal-integer-literal   hexadecimal-integer-literal   octal-integer-literal
 decimal-integer-literal_LL   hexadecimal-integer-literal_LL
 octal-integer-literal_LL

isnullFunction :
 isnull { handleOrSelf . attributeName }


literal :
 stringLiteral   charLiteral   integerLiteral   floatLiteral   booleanLiteral
```

```
locationSpec :
 identifier   integerLiteral

loopControlStatement :
 break   continue

lval :
 { ( ++   -- ) } quotedIdentifier arrayList

lvalStatement :
 lval { assignmentExpression }

methodName :
 quotedIdentifier

methodOrVar :
 handleOrSelf . attributeName (. attributeName )* ( paramList   arrayList )
 handle -> methodName { paramList }
 { ( &   * ) } { :: } ( quotedIdentifier :: )* identifier
     ( paramList   arrayList { ( ++   -- ) } )

multiplyingExpression :
 bitExpression ( ( *   /   % ) bitExpression )*

nameValue :
 attributeName : expression

nativeType :
 unsigned short   short   unsigned ( long   long long )
 long   long long   float   double   boolean   char
 wchar   octet   Object   any   string { < integerLiteral > }
 wstring { < integerLiteral > }   void

paramList :
 { paramSpec ( , paramSpec )* }

paramSpec :
 parameterName : expression   expression

parameterName :
 quotedIdentifier

primitiveExpression :
 literal   emptyFunction   isnullFunction   swalFunction   castExpression
 methodOrVar   expression
 sizeof ( quotedIdentifier   cType )
 new ( { :: } ( quotedIdentifier :: )* quotedIdentifier
 cType ) { ( [ arrayExpression ]   paramList ) }
 self

quotedIdentifier :
 identifier   string-literal

relateStatement :
 relate handleOrSelf relationshipSpec handleOrSelf { using handleOrSelf }

relationalExpression :
 shiftExpression ( comparisonOperator shiftExpression )*

relationshipName :
 quotedIdentifier

relationshipSpec :
 relationshipName

returnStatement :
 return { expression }
```

```
scopedMethodName :
 ( quotedIdentifier :: )+ quotedIdentifier

scopedType :
 { :: } ( quotedIdentifier :: )* quotedIdentifier

selectStatement :
 select { distinct className } handle from
  ( handleOrSelf extendedChain { where whereSpec }
     singleton className
     className ( where whereSpec   usingClause )
   )

setSpec :
 className   handleOrSelf extendedChain

shiftExpression :
 addingExpression ( ( <<   >> ) addingExpression )*

shiftOp :
 <<   >>

signExpression :
 { + - } unaryExpression

spawnStatement :
 spawn scopedType { paramSpec ( , paramSpec )* }

statement :
 declareStatement   createStatement ;   deleteStatement ;
 relateStatement ;   unrelateStatement ;   selectStatement ;   forStatement
 ifStatement   whileStatement   tryStatement   throwStatement ;
 loopControlStatement ;   transactionStatement ;   assertStatement ;
 spawnStatement ;   genAssignDeclareStatement ;   userMethodStatement ;
 superStatement ; lvalStatement ;   returnStatement ;   externStatement ;   statementBlock


statementBlock :
 { ( statement )* }

stringLiteral :
 ( string-literal )+

superStatement :
 super. methodName { paramList }

swalFunction :
 cardinalityFunction   inFunction

throwStatement :
 throw handle

transactionStatement :
 ( begin   commit   abort ) transaction

tryStatement :
 try statementBlock ( catchStatement )+

type :
 constDeclare ( nativeType   userDefinedType )

unaryExpression :
 { unaryOperator } primitiveExpression

unaryOperator :
 !   -   +   ~
```

```
unrelateStatement :
 unrelate handleOrSelf relationshipSpec handleOrSelf { using handleOrSelf }

userDefinedType :
 scopedType

userDereference :
 { ( ( * )+   & ) }

userMethodStatement :
 handle ( -> methodName )+ { paramList } { assignmentExpression }

usingClause :
 using className where whereSpec
  { readlock | writelock | nolock }
  { on empty create { values nameValue ( , nameValue )* } }

valueSpec :
 attributeName : expression   chainSpec : expression

whereExpression :
 ( whereSpec   wherePrimitive )

wherePrimitive :
 ( handleOrSelf . attributeName arrayList   handleOrSelf )
    comparisonOperator primitiveExpression

whereSpec :
 whereExpression ( booleanOperator whereExpression )*

whileStatement :
 while booleanExpression statementBlock
```

# 5

# Build specification reference

This chapter provides a reference for the keywords and commands you use to compose and execute a Kabira Infrastructure Server (KIS) build specification. These appear in alphabetical order. Each page includes the syntax of the keyword or command, a discussion of usage, and an example.

## A simple example

The example below shows a very simple build specification. For a full build specification that uses all the standard features, see the section called "Complete build specification example" on page 114.

```
component Component1
{
        source MyIDLosFile.soc
        import ImportedComponent
        {
                importPath=/this/is/where/the/component/is;
        };
        library=someLibraryThisComponentNeeds;

        package Package1;
        package ::Package2
        adapter sybase
        {
                entity Package1::PersistentEntity
        };
};
```

## Understanding the syntax notation

The syntax description for each keyword or command uses the following conventions:

**bold** screen font indicates a literal item that appears exactly as shown

*italics* indicates a language element that is defined elsewhere

`plain text` indicates a feature of the notation, as follows:

( item ) parentheses group items together

( item )+ the item in parentheses appears one or more times

{ item } items in braces are optional

# adapter

Defines an adapter block.

## Syntax

```
adapter adapterName { buildLanguage };
```

## Description

`adapterName` is the name of a valid, installed service adapter.

An adapter block defines a service adapter for a specific model element. Each adapter must contain an interface or entity, depending on its type.

Adapters also have properties that affect they way they operate. Refer to the documentation for a specific service adapter for more information on the properties used by that adapter.

## Example

```
adapter corba
{
        interface ::MyPackage::anInterfaceA;
        interface ::MyPackage::anInterfaceB;
};
```

# component

Defines a component block.

## Syntax

```
component identifier  {implementation=service-name}
   { buildLanguage };
```

## Description

A component block defines a component being built and will contain all the information needed to build the component. A component can be implemented with IDLos, Java, or a mixture of both.

A component containing IDLos may implement one or more packages, or may simply contain interfaces to an external implementation. All packages in a given component must be listed in the order of dependency.

A component can act as a wrapper around a service implemented in another technology. The generated wrapper will behave just like any other component except that it uses a foreign implementation.

*`serviceName`* is the name of the foreign service being implemented.

Refer to the documentation for the individual adapter generator for more information on wrapping a foreign service.

# Properties

The valid component properties are:

- `importPath`
- `buildPath`
- `buildType`
- `cFlags`
- `ccFlags`
- `classPath`
- `debug`
- `timestamp`
- `includePath`
- `javacOptions`
- `javaPackageRoot`
- `ldFlags`
- `libraryPath`
- `library`
- `methodsPerFile`
- `numParallelCompiles`
- `name`
- `description`
- `engineGroup`
- `engineService`
- `extraArchiveFile`
- `undef`
- `placeSourceInArchive`

## Warnings

None

## Example

```
component MyComponent
{
    source MyIDLos.soc;
    package MyPackage;
};
component AnExternalComponent implementation=corba
{
        idlPath=MyIDL.idl;
};
```

# import

Specifies a component dependency, or aggregates components.

## Syntax

```
import identifier;
```

## Description

A component may depend on the services provided by another component. You import a component so that its interface definitions and link libraries can be loaded into the Design Center server at build time.

The Design Center always knows where to locate built-in KIS components. If you want to import your own components from an arbitrary location, you can specify this location using the `importPath` property. Note that this property follows normal scoping rules, so you can use different import paths for different components.

## Example

```
component MyComponent
{
    // set the include path to the component location
    importPath = /some/include/path;
    import DependentComponentA
};
```

# Macros

Allows you to substitute parameters from the command line.

## Syntax

Use in the build spec: `$(` macro-name `)`

Definition on the command line: `swbuild -o` *`macro-name`*`=` *`value`*

## Description

Before the build specification is parsed, macros are evaluated and substituted with values from the command line. Forms that do not correspond to the syntax are not macros, and no substitution takes place. To explicitly escape the `$(` sequence, you can use `$$(` . Thus `$$(NOTAMACRO)` becomes `$(NOTAMACRO)` after substitution. Because macros are evaluated before parsing, they can be used anywhere:

```
component $(COMPONENT_NAME)
{
        name=$(VALUE);
        $(NAME)=$(VALUE);
        name="Macros can even be in $(STRINGS)";
};
```

An undefined macro evaluates to the name of the macro. For example, if `$(VALUE)` is not defined, it evaluates to `VALUE` , and the Design Center issues a warning.

## Example

A typical use of macros is to pass property values from the command line.

```
component dcapiplugin
{
         source=dcapi.soc;
         package dcapi;

         includePath=../include;
         buildType=$(BUILD);
         numParallelCompiles=$(MAXPROCESS);
};
```

The command line associated with the example above would be something like:

```
swbuild -o BUILD=PRODUCTION -o MAXPROCESS=4
```

## Warnings

Macros are expanded anywhere, even in comments. This means you can't simply comment out lines containing undefined macros—you must remove the offending macro call.

## See also

the section called "swbuild" on page 113

# package

Specifies a package name.

## Syntax

```
package identifier;
```

## Description

*identifier* specifies the package name.

All packages to include in a component must be specified with a package statement.

## Properties

Properties may be associated with a package statement, as follows:

- javaPackageRoot

## Warnings

None.

## Example

```
component MyComponent
{
    //
    //    Default java package root for all packages unless explicitly overridden
    //    in the package statement
    //
    javaPackageRoot = kabira.platform;


    //
    //    Uses default Java package root of kabira.platform for Java bindings
    //
    package P1;

    //
    //     Use kabira.example as the package root for the Java bindings of package P2.
    //
    package P2
    {
         javaPackageRoot = kabira.example;
    };

    //
    //     Use kabira.snippet as the package root for the Java bindings of package P3.
    package P3
    {
         javaPackageRoot = kabira.snippet;
    };
};
```

# Properties

Specify options in a build specification.

## Syntax

```
propertyName = value;
```

# Description

Each property is a name-value pair that sets compiler build options for the containing component or adapter.

*propertyName* is the name of a defined build property from the properties table below.

*value* is a value as defined in the description column of the properties table below.

The following table is a complete list of the generic properties. A value of `Yes` in the `Global?` column indicates that the property can be specified outside of a component block in the file.

| Property | Type | Description | Global? |
|---|---|---|---|
| `importPath` | directory | Search paths used for imported components; default is "." | Yes |
| `buildPath` | directory | Directory for generated output; default is "." | Yes |
| `buildType` | choice | DEVELOPMENT (default) or PRODUCTION | Yes |
| `cFlags` | string | C compiler flags | Yes |
| `ccFlags` | string | C++ compiler flags | Yes |
| `classPath` | string | Add additional classes to the default build class path. The `classPath` property uses the JVM defined class path syntax, e.g. ":" separated JAR or class files. The default class path contains all dependent component JAR files. | No |
| `debug` | boolean | Enable action language debug code generation; default is false. | Yes |
| `document` | boolean | Enable documentation generation; default is `true`. | Yes |
| `timestamp` | choice | Enables time stamp collection on an action language, line by line basis; default is false. | Yes |
| `includePath` | directory | Search paths used for included files | Yes |
| `javacOptions` | string | Override the default javac options. | No. |
| `javaPackageRoot` | string | The package root to use for generated Java classes. The default value is `com.kabira.unclassified`. | No |
| `ldFlags` | string | Linker flags | Yes |
| `libraryPath` | directory | Search path for libraries | Yes |
| library | string | Libraries linked with runtime libraries | Yes |
| `loadClassAtVMStart` | string | Cause a Java class to be loaded when the JVM is started. The classes static initializer is executed. | No |
| `methodsPerFile` | numeric | Number of methods generated per file. Min: 10, Max: 100, default: 50 | Yes |
| `numParallelCompiles` | numeric | Number of parallel compilations; default is 1 | Yes |
| name | string | Name of the generated engine | No |
| `description` | string | Description string for this engine | No |
| `engineGroup` | string | Engine group for this component; default is "Applications" | No |

| Property | Type | Description | Global? |
|---|---|---|---|
| `engineService` | string | Add an engine service to this engine. | No |
| `extraArchiveFile` | file | Extra file to place in engine archive | No |
| `undef` | string | Inserts user-defined undef directives to avoid name conflicts with system headers | No |
| `placeSourceIn-Archive` | boolean | If true (default), the Design Center places the source files in component archives. | No |

When you use an adapter factory, additional properties specific to that adapter may also be available. See the relevant adapter factory documentation for these properties and the values they take.

## Example

```
// a global property will apply to both components
buildPath = some/build/path;
component MyComponentA
{
        // property local to MyComponentA
        includePath = some/include/path;
};
component MyComponentB
{
    // properties local to MyComponentB
    source = MyIDLosB.soc
    name=myWonderfulComponent
};
```

# source

Specifies a source file for a component.

## Syntax

```
source identifier;
```

## Description

*`identifier`* specifies the source filename.

Source files can be added to a component specification at a global or component level. A source file can be IDLos, action language, Java, or a C or C++ header file.

Source files must be listed in the correct order of dependency. It is best to list all source files, including `.act` files, in the specification instead of as a `#include` in the IDLos file. This will optimize Design Center server reloading.

## Properties

Properties may be associated with a source statement, as follows:

- `document`
- `includePath`

- `libraryPath`
- `loadClassAtVMStart` - only valid for Java source files.

## Warnings

None.

## Example

```
component MyComponent
{
    source /some/source/path/MyIDLos.soc;
    source /some/source/path/MyAL.act;
    source MyClass.java;    // A Java file - Java compiler is called

    //
    //     A Java class that will be loaded when the JVM starts
    //
    source Initialize.java
    {
         loadClassAtVMStart = a.b.c.Initialize;
    };
};
```

# swbuild

Build a component from the command line.

## Syntax

```
swbuild {-a} {-o macro-name=value} {specification-file}
```

## Description

The -a option suppresses building; the Design Center will perform an audit only.

The -o option allows you to set options for macros defined in the build specification.

The specification-file indicates the source file for the build specification. If you omit this argument, the Design Center reads the build specification from standard input.

## Example

```
swbuild mybuildspec.osc
```

## See also

the section called “Macros” on page 108

# Complete build grammar

```
adapterBlock  :
```

```
 {  (  propertyStatement     modelrefStatement  )*  }

adapterStatement  :
 adapter  identifier  {  adapterBlock  }  ;

componentBlock  :
 {  (  sourceStatement     importStatement     propertyStatement
     adapterStatement     modelrefStatement     includeStatement  )*  }

componentSpecification  :
 (  componentStatement     propertyStatement  )*

componentStatement  :
 component  identifier  (  implementation  identifier  )*
  {  componentBlock  }  ;

compoundQuotedLiteral  :
 quotedLiteral  (  quotedLiteral  )*

identifier  :
 [a-zA-Z][a-zA-Z0-9_]*

importStatement  :
 import  identifier  {  propertyBlock  }  ;

includeStatement  :
 include  (  <     "  )  filename (  "     >  )

modelrefStatement  :
 (  package     module     interface     entity  relationship
        operation     signal     attribute     role     key     entity  )
        scoped_name  {  propertyBlock  }  ;

nameToken  :
 identifier

propertyBlock  :
 {  (  propertyStatement  )*  }

propertyStatement  :
 nameToken  =  {  valueToken  }  ;

scoped_name  :
 {  ::  }  identifier  (  ::  identifier  )*

sourceStatement  :
 source  valueToken  {  propertyBlock  }  ;

unquotedLiteral  :
 literal-content

valueToken  :
 compoundQuotedLiteral     unquotedLiteral
```

# Complete build specification example

```
/*
 * Global properties.  These appear outside of any component
 * block, and apply to all components.
 *
 * The following tests global properties and tries to trick
 * our parser with macros and comments.
 *
 */

globalName1=globalValue1;
```

```
global$(NAME)2=global$(VALUE)2;
$(GLOBALNAME3)=$(GLOBALVALUE3);

/*comment*/global$(NAME)4/*$(comment)*/=global$(VALUE)4;

// Empty values are allowed. All three of the following are
// empty values:
emptyValue="";
emptyValue=;
emptyValue=$(EMPTYVALUE);

// A component statement describes a component to be built.
// In the plugin service project tree, this maps to ElemEngine

component Component1
{

    localName1=localValue1;
    local$(NAME)2=local$(VALUE)2;
    $(LOCALNAME3)=$(LOCALVALUE3);

    quotedValue = "

    What you see is what you get... almost.

    Escape sequences in this string follow the same
    rules as IDLos.

    So, you can \"escape a string\"
    insert a tab \t, etc.

    Check out unquoted literals if you are specifying file paths.

    ";

    escapedString="Escaped quote:\"  Newline:\n
                                     Escaped backslash:\\";

    quotedCompoundValue =
        "Notice how strings are allowed to "
        "continue, just like in C++;";

    quotedMacroValue = "quoted$(VALUE)1";

    // Note how unquoted literals don't have any escape sequences.
    // This makes file names on NT easy.

    pathValue1=/some/file;
    pathValue2=C:\some\file;

    escapedMacro1 = $$(NOTAMACRO);
    escapedMacro2 = "$$(NOTAMACRO)";
    escapedMacro3 = "This is $$(NOTAMACRO), OK?";

    // The source statement indicates a source file to be
    // loaded into the DC server.  Normally source files
    // do not need properties.  However, if you are using
    // #include in your source file, you may wish to define
    // an includePath which only applies to your file.  Be
    // aware that using #include is a bad idea.  You should
    // list all your files here (both IDLos and Action Language)
    // because it allows the DC to optmize reloads.

    source MyIDLosFile.soc
    {
        includePath = /some/path/name;
    };
```

```
    source MyActionLanguageFile.act;

    // The import statement establishes a dependency to
    // another component.  The DCAPI plugin will search for
    // this component using includePath.  You can specify
    // additional includePath, library, etc, parameters
    // for this component only by using a property block.

    import ImportedComponent
    {
        includePath=/this/is/where/the/component/is;
        library=someLibraryThisComponentNeeds;
    };

    // You must list the packages that will be
    // implemented in your componenent.  If leading
    // "::" is left off, it is implicit.  You can also
    // assign properties to model references.

    package Package1;
    package ::Package2
    {
        someproperty=somevalue;
    };

    // You may use the services of an adapter to
    // bind parts of your model to a particular
    // implementation.  It is up to each adapter as
    // to what sort of model references and properties
    // they allow

    adapter sybase
    {
       entity Package1::PersistentEntity
       {
           readString="Some SQL string";
           writeString="Some SQL string";
       };
    };

    // You can have more than one adapter, of course.

    adapter corba
    {
       interface /*comment*/ MySecondPackage::MyPublicInterface;
    };

};


// You can build multiple components in the same
// spec.  Componets can also have different implementations

component Component2 implementation java
{

        // blah, blah

};


// It should be possible to completely parameterize
// a build

component $(Component3)
{
```

```
    package $(Package3);
};
```

# 6

# PHP reference

This chapter contains reference pages for each Kabira Infrastructure Server (KIS) PHP extension function call, in alphabetical order. Each PHP extension function call appears in this section on its own page with a top-level syntax definition, a description of the function, and an example.

## About the notation

The syntax description for each statement or function uses the following conventions:

**bold** screen font indicates a literal item that appears exactly as shown

*italics* indicates a language element that is defined elsewhere

`plain text` indicates a feature of the notation, as follows:

( item ) parentheses group items together

( item )* the item in parentheses appears zero or more times

( item )+ the item in parentheses appears one or more times

{ item } items in braces are optional

( item1 | item2 ) the vertical bar indicates either item1 or item2 appears

For example:

```
keyword { optionalKeyword }
    { optionalThing }( zeroOrMoreRepeatingThing )*
    ( oneAlternative | otherAlternative | thirdAlternative )
```

# os_connect

Creates a new connection to KIS osw engine.

## Syntax

```
$conn_id =
os_connect ($host, $port, $username, $password, $mechanism, $mechanismdata);
$conn_id =
os_connect ($host, $port);
$conn_id =
os_connect ($host);
$conn_id =
os_connect ( );
```

## Description

This function creates a connection between the PHP process and the osw engine.

conn_id is a PHP variable.

host and port are optional parameters that locate the Web Adapter engine.

If host and port are not set they will default to the host and port specified in the `php.ini` file.

username, password, mechanism, and mechanismdata are optional authentication parameters. They must be specified if the Web Adapter has secure access enabled.

## Warnings

None

## Error Conditions

- Not able to connect to KIS server

# os_create

Creates a KIS object and returns its reference.

## Syntax

```
$objr =
os_create ($conn_id, $scopedName, $attrList);
$objr =
os_create ($conn_id, $scopedName);
```

## Description

This function creates an object and returns its reference in the variable objr. If the object has attributes, an optional attrList could be passed in to set the attribute initial values.

When the scopedName type is a singleton, this call acts like the create singleton in action language. It creates the singleton if it does not exist. Otherwise, it returns the object reference.

conn_id is the value returned by `os_connect()` .

objr is the return value.

scopedName is a fully scoped KIS interface name.

attrList is an optional associative array of attribute names and values.

## Example

```
//
// create a sales person with “name” initialized
//
$salesType = “myPackage::Salesman”;
$salesAttrArray = array(‘name’ => ‘Slick Willy’);
$salesHandle = os_create($conn_id, $salesType, $salesAttrArray);

//
// create a customer - without attribute array
//
$custType = “myPackage::Customer”;
$custHandle = os_create($conn_id, $custType);
```

## Error Conditions

- Not able to connect to KIS server
- Invalid scoped name
- Invalid attribute name
- Unsupported type
- No create access
- Duplicate Key

# os_delete

Deletes a KIS object.

## Syntax

```
os_delete($conn_id, $objrList);

os_delete($conn_id, $objr);
```

## Description

The second parameter may be a either a single object reference or an array containing a list of object references.

conn_id is the value returned by `os_connect()` .

objrList is an array of valid object instances to delete.

objr is a valid object instances to delete.

## Warnings

None.

## Example

```
//
// Delete all Orders
//
$sn = “mypackage::Order”;
$orderList = os_extent($conn_id, $sn);
os_delete($conn_id,$orderList);

//
// delete a single object
//
$objr = os_create($conn_id, $sn);
os_delete($conn_id, $objr);
```

## Error Conditions

- Not able to connect to KIS server
- Invalid handle
- No delete access

# os_disconnect

Disconnect from the KIS engine.

## Syntax

```
os_disconnect ($conn_id);
```

## Description

conn_id is the return value from a call to `os_connect()` .

## Warnings

None.

## Example

```
/* close a connection to the osw engine */
os_disconnect($xconn);
```

# os_extent

Retrieve the extent of object handles of a given type.

## Syntax

```
$objrList =
os_extent($conn_id, $scopedName, $attrList);
$objrList =
os_extent($conn_id, $scopedName);
```

## Description

This function will select objects of a given type. If attrList is provided, it contains a list of name-value pairs that are ANDed together as a where clause to filter the number of object handles being returned.

If the objects are keyed and attrList has key coverage, a keyed lookup will be performed.

conn_id is the value returned by `os_connect()` .

objrList is the return value and is a PHP array of scalar values.

scopedName is a string containing the fully scoped name of a type.

attrList is an optional associative array of attribute names and values.

## Warnings

None.

## Example

```
//
// get all customers
//
$sn = “myPackage::Customer”;
$custList = os_extent($conn_id, $sn);

//
// return all customers in California
//
$sn = “myPackage::Customer”;
$whereClause = array(‘state’ => ‘CA’);
$custList = os_extent($conn_id, $sn, $whereClause);
```

## Error Conditions

- Not able to connect to KIS server

- Invalid scoped name
- Invalid attribute name
- Unsupported type or bad value
- No extent access

# os_get_attr

Retrieves the values of the attributes in a KIS object.

## Syntax

```
$attrList =
os_get_attr($conn_id, $objr, $filter);
$attrList =
os_get_attr($conn_id, $objr);
```

## Description

This function retrieves attribute values from an object. If filter exists, only the values of those attributes named in the filter array will be returned. Otherwise, all attributes values will be returned.

conn_id is the value returned by `os_connect()` .

attrList is an optional associative array of attribute names and values.

objr is a KIS object handle.

filter is an optional associative array of attribute names and values.

## Warnings

None.

## Example

```
//
// get all attributes of a cusotmer
//
$attrs = os_get_attr($conn_id, $custHandle);
for  ( reset($attrs);  $name = key($attrs); next($attrs ) )
{
     $avalue = $attrs[$name];
     print “$name = $value\n”;
}

//
// get the state attribute only
//
$filter = array ( “state” => “” );
$attrs = os_get_attr($conn_id, $custHandle, $filter);
$state = $attrs[“state”];
print “This customer is in state of $state\n”;
```

## Error Conditions

- Not able to connect to KIS server
- Invalid handle
- Invalid attribute name

# os_invoke

Invoke an operation on a KIS object.

## Syntax

```
$returnValue =
os_invoke($conn_id, $objr, $opName, $param, $ex);
$returnValue =
os_invoke($conn_id, $objr, $opName, $param);
$returnValue =
os_invoke($conn_id, $objr, $opName);
```

## Description

This function is used to invoke an operation on an object. The returnValue is not required on void operations. If a parameter is an inout or out parameter the value of that array element in param will be modified with the result parameter value after a call. If the operation raises user defined exception, $ex will contain the name of the exception type as a string upon return.

conn_id is the value returned by `os_connect()` .

returnValue is the return value of the operation upon completion.

objr is a valid object instance handle.

opName is the name of an operation.

param is a nested associative array of parameter name and value pairs.

ex is the name of user exception thrown by the operation.

## Example

```
//
// call a void operation that does not have any params
//
os_invoke($conn_id, $objr, “runtest”);

//
// call an operation with parameters that returns a boolean
//
$params = array (“name” => “Smith”, “number” => 5);
```

```
$ret = os_invoke($conn_id, $objr, “register”, $params);
if ($ret)
{
 print “OK\n”;
}
else
{
 print “register failed\n”;
}
```

When an operation raises user defined exceptions, os_invoke() can get the exception type, but not the exception data if it has member fields.

```
//
//  operation with user defined exceptions
//
$userex = ““;
$params = array();
os_invoke($conn_id, $objr, “myOp”, $params, $userex);
if ($userex != ““)
{
 print “Caught user exception $userex\n”;
}
```

Array and sequence types can be used as in, out, inout parameters and return values.

```
//
//  array or sequence type as out param
//
$params = array ( “myList” => array() );
$ret = os_invoke($conn_id, $objr, “getList”, $params);
$list = $params[“myList”];
for ($i = 0; $i < count($list); $i++)
{
 $value = $list[$i];
 print “list[ $i ] = $value\n”;
}
```

## Error Conditions

- Not able to connect to KIS server
- Invalid handle
- Invalid operation name
- Invalid parameter name
- Unsupported type or bad value
- Application exception

# os_relate

Relates two interfaces.

## Syntax

```
os_relate($conn_id, $fromObjr, $roleName, $toObjr);
```

## Description

This function relates two objects using the relationship role roleName.

conn_id is the value returned by `os_connect()` .

fromObjr is a valid object reference handle.

roleName is the name of a role in a relationship between the “from” and “to” objects.

toObjr is a valid object reference handle.

## Warnings

None.

## Example

```
$salesType = “myPackage::Salesman”;
$salesAttrArray = array(‘name’ => ‘Slick Willy’);
$salesHandle = os_create($conn_id,$salesType, $salesAttrArray);

$custType = “myPackage::Customer”;
$custAttrArray = array( ‘name’ => ‘Lucent’, ‘state’ => ‘NJ’);
$custHandle = os_create($conn_id,$custType, $custAttrArray);
os_relate($conn_id,$salesHandle,  “hasCust”, $custHandle);
```

## Error Conditions

- Not able to connect to KIS server
- Invalid handle
- Invalid role name

# os_role

Retrieves an array of handles by navigating across a relationship.

## Syntax

```
$objrList =
os_role($conn_id, $objr,  $roleNam, $attrList);
$objrList =
os_role($conn_id, $objr,  $roleNam);
```

## Description

This function returns an array of object references as it navigates across the appropriate relationship. The name-value pairs in attrList are ANDed together to act as a where clause to filter the number of object handles being returned.

conn_id is the value returned by `os_connect()` .

objrList is a list of object references.

objr is a KIS object handle.

roleName is a fully scoped relationship role name.

attrList is an optional associative array of attribute names and values.

## Warnings

None.

## Example

```
/*
** assume Salesperson is related 1:M with customer
** and that $salesHandle is already populated with a
** valid Salesperson.
*/
$cList = os_role($conn_id, $salesHandle, “packageName::relationshipName::sellsTo”);
for ( reset($cList);  $cust = current($cList); next($ctList))
{
     /* do something with $cust */
}
```

## Error Conditions

- Not able to connect to KIS server
- Invalid handle
- Invalid role name
- Invalid attribute name
- Unsupported type or bad value

# os_set_attr

Sets a value in an attribute of a KIS object.

## Syntax

```
os_set_attr($conn_id, $objr, $attrList);
os_set_attr($conn_id, $objr, $name, $value);
```

## Description

This function sets a value of an object attribute. More than one attribute value can be set in an object.

conn_id is the value returned by `os_connect()` .

objr is a KIS object handle.

attrList is an optional associative array of attribute names and values.

## Example

```
//
// set the name of a customer
//
$attrArray = array('name' => 'John');
os_set_attr($conn_id, $objr, $attrArray);

//
// or using
//
os_set_attr($conn_id, $objr, "name", "John");

//
// set array attribute
//
$value = array (1, 2, 3, 4, 5);
$attrArray = array ("longList" => $value);
os_set_attr($conn_id, $objr, $attrArray);
```

## Error Conditions

- Not able to connect to KIS server
- Invalid handle
- Invalid attribute name
- Attribute is readonly
- Unsupported type or bad value
- Duplicate Key

# os_unrelate

Unrelates two objects.

## Syntax

```
os_unrelate($conn_id, $fromObjr, $roleName, $toObjr);
```

# Description

This function unrelates two objects that are currently related using the relationship role roleName.

conn_id is the value returned by `os_connect()` .

fromObjr is a valid object reference handle.

roleName is a relationship role name.

toObjr is a valid object reference handle.

# Warnings

None.

# Example

```
$custList = os_role($conn_id, $salesHandle,  ":sellsTo");
for (reset($custList); $cust = current($custList); next($custList))
{
     os_unrelate($conn_id, $salesHandle,  'hasCust', $cust);
}
```

# Error Conditions

- Not able to connect to KIS server
- Invalid handle
- Invalid role name

# 7

# String operations

This reference chapter describes the Kabira Infrastructure Server (KIS) built-in string operations. The operations are presented in the following groups:



## Copy and append operations

The operations in this group allow you to copy and append values to a string.

### stringCopy()

Copies the string argument to a string.

Syntax

```
void stringCopy(in string src)
```

Exceptions

```
None
```

# stringAppend()

Construct a string type from the argument, if not already a string, and append it to the string. There is a version for each supported type:

Syntax

```
void stringAppend(in string src)
void stringAppend(in char[] src)
void stringAppend(in short value)
void stringAppend(in long value)
void stringAppend(in unsigned short value)
void stringAppend(in unsigned long value)
void stringAppend(in boolean value)
void stringAppend(in long long value)
void stringAppend(in unsigned long long value)
void stringAppend(in float value)
void stringAppend(in double value)
void stringAppend(in char value)
void stringAppend(in octet value)
```

Exceptions

```
None
```

# formatAppend()

Appends the value to the string. These operations allow you specify a format (width and precision) for the value being appended to the string.

The supported types are:

- double
- long
- string
- unsigned long

Syntax

```
void formatAppend(
   in long width,
   in long precision,
   double value)
void formatAppend(
   in long width,
   in long precision,
   in unsigned long value)
void formatAppend(
   in long width,
   in long precision,
   in unsigned long long value)
void formatAppend(
   in long width,
   in long precision,
   in long value)
void formatAppend(
   in long width,
```

```
   in long precision,
   in string value)
```

Exceptions

```
None
```

## formatHexAppend()

Appends a hexadecimal value to a string. This operation allows you to specify a format (width and precision) for the value being appending.

Syntax

```
void formatHexAppend(
   in long width,
   in long precision,
   in unsigned long value)
```

Exceptions

```
None
```

# Comparison operations

The operations in this group allows you to compare a string with another lexicographically.

## stringCompare()

Compares the string with the string argument. The value returned indicates the result of the lexicographical comparison:

| Return value | Significance |
|---|---|
| >0 | string > argument |
| 0 | string == argument |
| <0 | string < argument |

Syntax

```
long long stringCompare(in string string)
```

Exceptions

```
None
```

## stringNCompare()

Compares the string with the string argument for the specified number of bytes. The value returned indicates the result of the lexicographical comparison:

| Return value | Significance |
|---|---|
| >0 | string > argument |
| 0 | string == argument |

| Return value | Significance |
|---|---|
| <0 | string < argument |

Syntax

```
long long stringNCompare(in string string, in long long length)
```

Exceptions

```
None
```

# Index operations

The operations in this group allow you to find the position of a substring or character, determine whether a substring or character is contained in the string or whether the string ends or begins with a substring.

## indexOf()

Returns the index of the first occurrence of the substring if found; otherwise -1.

Syntax

```
long long indexOf(in string substr)
```

Exceptions

```
None
```

## lastIndexOf()

Returns the index of the last occurrence of the substring, if found; otherwise -1.

Syntax

```
in long long lastIndexOf(in string substr)
```

Exceptions

```
None
```

## indexOf()

Returns the index of the first occurrence of the character, if found; otherwise -1.

Syntax

```
long long indexOf(in char token)
```

Exceptions

```
None
```

## lastIndexOf()

Returns the index of the last occurrence of the character, if found; otherwise -1.

Syntax

```
long long lastIndexOf(in char token)
```

Exceptions

```
None
```

## stringContains()

Indicates whether the string argument is contained in the string.

Syntax

```
boolean stringContains(in string string)
```

Exceptions

```
None
```

## stringContains()

Indicates whether the char argument is contained in the string.

```
boolean stringContains(in char token)
```

Exceptions

```
None
```

## beginsWith()

Indicates whether the string begins with the string argument.

Syntax

```
boolean beginsWith(in string substr)
```

Exceptions

```
None
```

## endsWith()

Indicates whether the string ends with the string argument.

Syntax

```
boolean endsWith(in string substr)
```

Exceptions

```
None
```

# Type conversion operations

The operations in this group allow you to construct a non-string type from a string.

Some of the conversion operations accept an argument of type NumericBase. NumericBase is an enum defined in the string type.

```
enum NumericBase
{
   BaseDerived = 0,
   Base8 = 8,
   Base10 = 10,
   Base16 = 16
};
```

Use the following values to specify the corresponding base:

- `SWString::BaseDerived`
- `SWString::Base8`
- `SWString::Base10`
- `SWString::Base16`

The default is `SWString::BaseDerived` , which causes the function to determine the base from the string value:

- decimal constant begins with a non-zero digit, and consists of a sequence of decimal digits
- octal constant consists of the prefix 0 optionally followed by a sequence of the digits 0 to 7 only
- hexadecimal constant consists of the prefix 0x or 0X followed by a sequence of the decimal digits and letters a (or A) to f (or F) with values 10 to 15 respectively.

## stringToChar()

Returns the string value converted to a char.

Syntax

```
char stringToChar(void)
```

Exceptions

```
swbuiltin::ExceptionDataError
```

## stringToBoolean()

Returns the string value converted to a boolean.

Syntax

```
boolean stringToBoolean(void)
```

Exceptions

```
swbuiltin::ExceptionDataError
```

# stringToOctet()

Returns the string value converted to a octet.

Syntax

```
octet stringToOctet(in NumericBase base)
```

Exceptions

```
swbuiltin::ExceptionDataError
```

# stringToShort()

Returns the string value converted to a short.

Syntax

```
short stringToShort(in NumericBase base)
```

Exceptions

```
swbuiltin::ExceptionDataError
```

# stringToUShort()

Returns the string value converted to a unsigned short.

Syntax

```
unsigned short stringToUShort(in NumericBase base)
```

Exceptions

```
swbuiltin::ExceptionDataError
```

# stringToLong()

Returns the string value converted to a long.

```
long long stringToLong(in NumericBase base)
```

Exceptions

```
DSEBuiltin::ExceptionDataError
```

# stringToULong()

Returns the string value converted to a unsigned long.

```
unsigned long stringToULong(in NumericBase base)
```

Exceptions

```
swbuiltin::ExceptionDataError
```

## stringToLongLong()

Returns the string value converted to a long long.

Syntax

```
long long stringToLongLong(in NumericBase base)
```

Exceptions

```
swbuiltin::ExceptionDataError
```

## stringToULongLong()

Returns the string value converted to a unsigned long long.

Syntax

```
unsigned long long stringToULongLong(
   in NumericBase base)
```

Exceptions

```
swbuiltin::ExceptionDataError
```

## stringToSize()

Returns the string value converted to a SW_SIZE.

Syntax

```
SW_SIZE stringToSize(in NumericBase base)
```

Exceptions

```
swbuiltin::ExceptionDataError
```

## stringToFloat()

Returns the string value converted to a float.

Syntax

```
float stringToFloat(void)
```

Exceptions

```
swbuiltin::ExceptionDataError
```

## stringToDouble()

Returns the string value converted to a double.

Syntax

```
double stringToDouble(void)
```

Exceptions

```
swbuiltin::ExceptionDataError
```

# String manipulation operations

The operations in this group allow you to manipulate the contents of a string.

## replaceChar()

Replaces occurrences of character `oldChar` with `newChar`

Syntax

```
void replaceChar(in char oldChar, in char newChar)
```

Exceptions

```
None
```

## replaceCharAt()

Replaces character at index `pos` with `newChar` .

Syntax

```
void replaceCharAt(in long long pos, in char newChar)
```

Exceptions

```
ExceptionArrayBounds
```

## replaceString()

Replaces occurrences of `oldString` with `newString` .

Syntax

```
void replaceString(in string oldString, in string newString)
```

Exceptions

```
None
```

## replaceStringAt()

Replaces the substring from `start` to `end` positions with `newString` .

Syntax

```
void replaceStringAt(
   in long long start,
   in long long end,
   in string newString)
```

Exceptions

```
ExceptionArrayBounds
```

## insertChar()

Inserts the character `data` at the index `pos`.

Syntax

```
void insertChar(in long long pos, in char data)
```

Exceptions

```
ExceptionArrayBounds
```

## insertString()

Inserts the string `data` at the index `pos`.

Syntax

```
void insertString(in long long pos, in string data)
```

Exceptions

```
ExceptionArrayBounds
```

## substring()

Returns the substring from position `start` to position `end`.

Syntax

```
string substring(in long long start, in long long end)
```

Exceptions

```
ExceptionArrayBounds
```

## trim()

Removes any leading and trailing whitespace from the string.

Syntax

```
void trim(void)
```

Exceptions

```
None
```

## remove()

Removes the substring starting at position `start` and ending at position `end` from the string.

Syntax

```
void remove(in long long start, in long long end)
```

Exceptions

```
ExceptionArrayBounds
```

### pad()

Appends a number of `data` characters to the string so that the total number of characters is `totalLength` .

Syntax

```
void pad(in long long totalLength, in char data)
```

Exceptions

```
ExceptionArrayBounds (totalLength<0)
```

## Case conversion operations

### toupper()

Returns a string constructed by converting all characters to uppercase.

Syntax

```
string toupper(void)
```

Exceptions

```
None
```

### tolower()

Returns a string constructed by converting all characters to lowercase.

Syntax

```
string tolower(void)
```

Exceptions

```
None
```

## Diagnostic and initialization operations

The operations in this group allow you to valid, initialize, and set the size of strings.

### stringValid()

Indicates whether the string is valid.

Syntax

```
boolean stringValid(in long long maxLength)
```

Exceptions

```
None
```

## reset()

Initializes the string.

Syntax

```
void reset(void)
```

Exceptions

```
None
```

## resetSize()

Preallocates a buffer of the size `maxStringSize` .

Syntax

```
void resetSize(in long long maxStringSize)
```

Exceptions

```
None
```

# C-type string operations

The operations in this group allow you to obtain a C-type string from a string or append or copy a C-type string to a string.

## getCString()

Returns a pointer to a null terminated C-type string.

Syntax

```
char *getCString()
```

Exceptions

```
None
```

## attachConstBuffer()

Attach to a buffer which cannot be modified (typically shared memory).

Syntax

```
void attachConstBuffer(in char *buffer, in long long length)
```

Exceptions

```
None
```

## stringNCopy()

Copies a C-type string into a string object for the specified number of bytes

Syntax

```
void stringNCopy(SW_SIZE length, char *cString)
```

Exceptions

```
None
```

## stringNAppend()

Appends a C-type string to a string.

Syntax

```
void stringNAppend(SW_SIZE length, char *cString)
```

Exceptions

```
None
```

# 8

# IDLosDoc markup language

This chapter explains how to create component documentation by placing IDLosDoc tags in a component's source files. The documentation generated from these tags is delivered on line by the Kabira documentation server, which is instantiated when the Kabira Design Center runs.

# IDLosDoc tags

This section describes the tags used for IDLosDoc markup.

## Model documentation structured comment syntax

All structured comment blocks must start with the three-character anchor "/**" and end with the two-character sequence "*/". The text enclosed by these two anchors is considered to be embedded documentation. If an element has no structured comment, it will not be documented.

There are three kinds of tagged comments. One designates the semantic content of the comment, and is specified using an "@" symbol. The others allow the developer to specify structural and content for model documentation, and they conform to XML syntax.

## Semantic tag syntax

This section describes semantic tags. Semantically tagged comments are embedded in structured comment blocks, and have the following format:

```
/**
        @<tag name 1> [< name 1>] [<Description 1>]
        ...
        @<tag name n> [< name n>] [<Description n>]
*/
<code block associated with the structured comment>
```

The "@<...>" constructs are the actual tags used to define different kinds of item descriptions.

Each tag must begin on its own line and all leading white spaces are ignored. This tagged paragraph also includes any following lines up to, but not including, either the first line of the next tag or the end of the structured comment block as indicated by the "*/" anchor. The definition of the tagged item follows the "*/" anchor on the next line.

**Tag table** The following table lists the structured comment tags supported by the document system.

| Tag | Description | Constraints |
|---|---|---|
| `@abstract` | short description of the element | One per element |
| `@copyright` | copyright notice | One per package |
| `@deprecated` | marks the element as deprecated | One per element |
| `@discussion` | longer descriptive text | One per element |
| `@example` | code example. The `@example` tags are indexed as examples in the documentation. | None |
| `@exception` | describes an exception raised by the operation | Required; one per raised exception in operation |
| `@javaexample` | Java specific code example. | None |
| `@parameter` | operation parameter | Required; one per parameter in operation |
| `@preformatted` | preformatted text. All white space and formatting is preserved. | None |
| `@private` | do not generate IDLos documentation output for this element, or any elements contained within this element. | One per element |
| `@result` | operation return value | Required if operation has structured comment |
| `@see` | hyperlink that uses simplified code to identify the target within the document system. | None |
| `@warnings` | textual warnings | None |

**@abstract** The `@abstract` tag is an optional tag that precedes a documentable model element. It is used to provide a brief one sentence summary of the element. If no `@abstract` is provided, the first sentence of @discussion will be used. The usage:

```
/**
  @abstract IDLos static definitions for HTTP client.
 */
```

**@copyright** The `@copyright` tag is used to insert a copyright statement for package in the model documentation. If none is provided, the following line will be used:

Copyright © Kabira Technologies, Inc. All rights reserved.

The usage:

```
/**
  ...
  @copyright 2003, 2009 by Kabira Technologies, Inc. Confidential Property
  ...
*/
```

**@deprecated** The `@deprecated` tag is used to document an element that should no longer be used (though it may still work). The usage:

```
/**
@deprecated This function is replaced by this operation.
*/
```

**@discussion** The `@discussion` tag is an optional tag that is used to create general documentation for a model element. There can be only one @discussion tag per model element.

The usage:

```
/**
@discussion List of supported HTTP methods.
*/
enum MethodType
{
    /** HTTP OPTIONS method. */
    OptionsMethod,
    /** HTTP GET method. */
    GetMethod,
    /** HTTP HEAD method. */
    HeadMethod,
    /** HTTP POST method. */
    PostMethod,
    /** HTTP PUT method. */
    PutMethod,
    /** HTTP DELETE method. */
    DeleteMethod,
    /** HTTP TRACE method. */
    TraceMethod
};
```

**@example** The `@example` tag can be used to add code snippets to model element documentation. The white space and line breaks are preserved in the resulting output.

The usage:

```
/**
  @example
  entity E
  {
     attribute TimerHandle timerHandle;
  };
 */
```

**@exception** The `@exception` tag is used to document an exception raised by an operation.

The usage:

```
/**
 @discussion Your CPU has melted.
 */
exception Melt
{
  /** Read about your melted CPU */
  string message;
};
/**
 @discussion Your water has boiled.
 */
exception Boil
{
  /** Put your tea bag in the water */
```

```
  string message;
};
/**
*
@discussion This operation does something that results in a great deal of heat.
@exception Melt This is raised when the operation causes the CPU
to melt.
@exception Boil This is raised when the operation causes your tea
water to boil.
*/
void doSomething() raises (Melt, Boil);
```

**@javaexample** The `@javaexample` tag can be used to add Java code snippets to model element documentation. The white space and line breaks are preserved in the resulting output. This tag is useful for adding Java specific examples to IDLos components that are exposed with Java bindings.

The usage:

```
/**
  @javaexample
  public class MyJavaClass
  {
     int     value;
  };

*/
```

**@parameter** The `@parameter` tag documents a parameter in an operation. It is usually accompanied by a `@result` tag, which documents the return type for the operation.

Lacking either <name> or <description> is a syntax error and no documentation will be generated.

The usage:

```
/**
    @abstract    receive a line with timeout.
    @discussion
        Called to get a line (i.e. sequence of bytes
        delimited by CRLF) from the HTTP server.
    @parameter       timeout      time out value in seconds.
    @parameter       responseLine response line.
    @result
        can be one of
            OperationSucceeded,
            Disconnected,
            OperationTimedOut,
            OperationFailed
*/
Status recvLineTimeout(in  long   timeout,
                       out string responseLine);
```

**@preformatted** You can use the `@preformatted` tag to create preformatted text for your documentation. The whitespace and line breaks are preserved in the resulting output.

The usage:

```
/**
 ...
 @preformatted
   This is pre-formatted text:
     Here's one line.
        This one's indented further.
```

```
 ...
*/
```

**@private** The `@private` tag instructs the documentation system not to generate any output. This disabling of output applies to the element, and any elements contained by the element.

The usage:

```
/** @private */
interface interfaceNoDoc
{
 attribute char c;
};
```

**@result** The `@result` tag is used to document the result returned by an operation. For more information see `@parameter`.

**@see** The `@see` tag allows you to create the following 3 types of reference hyperlink to other documentation:

• the full URL

```
/**
 @see http://www.kabira.com
 */
```

• the other repository element in the SAME page

i.e. In package content, content::fields::Resolution can refer to content::ReserveResolution as follows:

```
/**
 @see ReserveResolution
 */
```

• the relative path or file names from $SW_HOME

```
/**
 @see distrib/kabira/kts/config/samples/
 */
```

or

```
/**
 @see distrib/kabira/kts/config/samples/sampleuserdata.xsd
 */
```

**@warnings** The `@warnings` tag is an optional tag that can be used to provide a description of any appropriate warnings. The usage:

```
/**
 @warnings This is a test for the warnings heading.
*/
```

**White space control** In structured comments, the system will convert two newlines (or an empty line) into a paragraph break

**Tag positioning** Semantic tags must be placed inside of structured comments immediately before the model element they describe, with the exception of `@parameter` , `@exception`, and `@result` tags, which are grouped in the structured comment block preceding the operation to which the parameters belong.

**Special tag character** To display @ in the documentation, use @@ to avoid the document system to treat it as an invalid tag.

**Special HTML characters** To display < and > in the documentation, one should use their HTML code as < and > accordingly to avoid them getting interpreted as HTML tags.

**XHTML tags support** The documentation system supports well-formed HTML tags. (XHTML implies well-formed HTML.) The definition of well-form is to have a pair of HTML tags such as

```
<pre></pre>
```

or

```
<p/>
```

**Left alignment** The documentation system will find the line with the fewest white spaces in the IDLos doc block and remove the number of white spaces from each line. For example,

```
package test
{
        /**
        @preformatted
        if a in plan
  put a tab in the line
          else if b in plan
   put 2 tabs in the line
        done
        */
        interface ifc
        :
```

This will generate the following text in the browser:

Preformatted:

```
if a in plan
 put a tab in the line
  else if b in plan
  put 2 tabs in the line
done
```

# Javadoc support

IDLosDoc in IDLos components exposed with Java bindings is translated to javadoc tags. See Table 8.1 on page 150 for the mapping between IDLosDoc and javadoc tags.

### Table 8.1. IDLosDoc Mapping to Javadoc

| IDLosDoc Tag | Javadoc Tag |
|---|---|
| @abstract | Tag stripped. Value passed through. |
| @copyright | Tag and value stripped. |
| @deprecrated | @deprecated |
| @discussion | Tag stripped. Value passed through. |

| | |
|---|---|
| `@example` | Tag and value stripped. |
| `@exception` | `@throws` |
| `@javaexample` | *`{@code text}`* |
| `@parameter` | `@param` |
| `@preformatted` | `{@literal text}` |
| `@private` | Tag and value stripped. |
| `@result` | `@return` |
| `@see` | `@see` |
| `@warnings` | Tag stripped. Value passed through prefixed by a `Warnings:` label. |

Any tags, not recognized as IDLosDoc tags are passed through unmodified to the generated Java bindings for a component. This provides full support for all supported javadoc tags.

# Example

Documentation is generated for all interfaces. This behavior can be changed used the `document` flag in a build specification.

```
component test
{
        source test.sdl;
        source test.soc;
        source test2.soc
        {
                document = true;
        };
        source test2.sdl
        {
                document = false;
        };
        package test;
};
```

# Example

```
/**
@abstract       Package test displays how one uses tags.
@discussion     We'll demonstrate how one can use tags like @@abstract,
          @@discussion, and so on.

One can also use the HTML tags as follows:
        <ol>
        <li>Item 1</li>
        <li>Item 2</li>
        <li>Item 3</li>
        </ol>

                This is a complete set of tags supported by package.
                The &lt;pre&gt; HTML tag is supported now.
                <pre>
        NOTE:
        -----
        It reserves all the white space at the beginning of the line.
          </pre>
@copyright      Copyright of Kabira Technologies, Inc.
@warnings       This is the 1st warnings.
```

```
@example
        You can do something like the followings:

    test::negativeFoo = 12;
    test:typedefString = "testing";

@preformatted
        The first @@preformatted tag.
        It preserves all the white space as you can see from
        the generated page.
@warnings       This is the 2nd warnings.
@warnings       This is the 3rd warnings.
@see            http://www.kabira.com
@see            interfaceBar
@see            distrib/kabira/kts/config/samples
@see            distrib/kabira/kts/config/samples/sampleuserdata.xsd
@example
A 2nd example.
@preformatted
                The 2nd preformatted tag to
        demonstrate that you can have more than
  one preformatted tag.
*/

package test
{
        /**
         * @discussion A const string named constString.
         */
        const string constString = "<STRING>";

        /**
                @abstract A typedef string named typedefString.
        */
        typedef string typedefString;

        /**
   A sequence typdef of long named typedefSeqLong.
        */
        typedef sequence<long> typedefSeqLong;

        /** @abstract A native SW_INT32 named nativeFoo. */
        native SW_INT32 *nativeFoo;

        entity bar;
        interface interfaceFoo;

        /** An exception named ExceptionFooError */
        exception ExceptionFooError
        {
                /** An message for exception ExceptionFooError. */
                string messageFoo;
        };

        /**
         @discussion
              A structure is a very useful thing.
         */
        struct FooCase
        {
                /** A string label */
                string    label;
                /** An object element */
                Object    element;
        };

        /**
```

```
        @abstract       An enum named Color.
        @discussion     This enum is used as the input selection
                        for unionColor.
*/
enum Color
{
        /** The color Red.*/
        Red,
        /** The color White.*/
        White,
        Blue
};

/**
 @abstract     A union named unionColor.
 @discussion  This switch is based on enum Color.
 @warnings     Only valid colors are Red, White, and Blue.
*/
union unionColor switch(Color)
{
        /**
         @abstract      Color Red is a short red.
        */
        case Red:
                short red;
        /**
         @discussion    Color White is a long white.
        */
       case White:
                long white;
        /** Color Blue is a string blue.*/
        case Blue:
                string blue;
};

/**
 @abstract    A module which is like a package except no
copyright.
 @discussion  This is module moduleTest in package test.
 @deprecated   This should not be used any more.
 */
module moduleA
{
        /** a const in moduleA */
        const char constA = 'a';

        /** @private */
        const short constModuleAPrivate = 42;

        /** moduleB to test the nested module */
        module moduleB
        {
                /**
                @abstract An entity named entTest inside
of module.
                */
                entity entTest
                {
                        /** an attribute long named attLong */
                        attribute long attLong;
                };
        };
};

entity foo
{
        attribute boolean groupBroadcast;
```

```
        attribute string fooPrivateStringAttr;
        void setGroupBroadcast(in boolean b);
};

entity bar
{
        attribute long a;
        key Primary { a };
        long setOperationWithException(in  string s,
   out long times)
                raises(ExceptionFooError);
};

/**
 @discussion An interface named interfaceFoo. */
interface interfaceFoo
{
        /** const in interfaceFoo */
        const unsigned long constIfc = '\0';

        /**
         @abstract Set the groupBroadcast.
         @parameter b   Turn on/off the groupBroadcast.
        */
        void setGroupBroadcast(in boolean b);
};
expose entity foo with interface interfaceFoo;

/** @abstract An interface named interfaceBar */
interface interfaceBar
{
        /**
        @attribute a a legacy tag @@attribute a
        @abstract interfaceBar's attribute a */
        attribute long a;

        /**
         @discussion a primary key of attribute a.
         */
        key Primary { a };

        /**
         @discussion  The discussion for this operation.
         @parameter   s             a string
         @parameter   times   number of times to have bar
         @result      number of foo exceptions
         @exception   ExceptionFooError an exception food error
         @example
         //
         // Action language
         //
         declare string  s;
         declare long    times;
         declare long    count;
         declare ExceptionFooError efe;

         count = setOperationWithException(s, times)
                 raises(efe);
        */
        long setOperationWithException(in string s, out long times)
                raises(ExceptionFooError);
};
expose entity bar with interface interfaceBar;

/**
   @abstract This is relationship FooBarRelationship
  @example
```

```
                relationship FooBarRelationship
                {
                        role foo FooToBar 0..* bar;
                        role bar BarToFoo 1..1 foo;
                };
         */
        relationship FooBarRelationship
        {
                /** @abstract a role named FooToBar
                @discussion role FooToBar maps from foo to bar
                */
                role foo FooToBar 0..* bar;

                /** @abstract a role named BarToFoo
                @discussion role BarToFoo maps from bar to foo
                */
                role bar BarToFoo 1..1 foo;
        };

        /** @private */
        const long JohnSilver = 8;

        /** @private */
        interface MYOB
        {
                attribute char c;
        };
        entity MYOBImpl
        {
                attribute char c;
        };
        expose entity MYOBImpl with interface MYOB;

        action test::foo::setGroupBroadcast
        {`
                self.groupBroadcast = b;
        `};

        action test::bar::setOperationWithException
        {`
                return 1234;
        `};
};
```

# Index

## F



## I



## K



## L



## M



## N

## T



## U



## V



## W